



Kodo™ 4.0.0 Developers Guide for JPA/JDO

Copyright © 2001 - 2006 BEA Systems Inc.

Kodo™ 4.0.0 Developers Guide for JPA/JDO

Copyright © 2001 - 2006 BEA Systems Inc.

Table of Contents

1. Introduction	1
1. Kodo JPA/JDO	3
1.1. About This Document	3
1.2. Sales Inquiries	3
2. Kodo Installation	4
2.1. Overview	4
2.2. Installation	4
2.2.1. Updates	4
2.2.2. Key Files in the Download	4
2.2.3. Quick Start	4
2.2.4. Upgrading from Previous Kodo Versions	4
2.2.5. Requirements	5
2.2.6. Terminology	5
2.2.7. Windows Installation	5
2.2.8. POSIX (Linux, Solaris, Mac OS X, Windows with cygwin, etc.) Installation	5
2.2.9. Common Installation Problems	6
2.2.10. Resources	6
2. Java Persistence API	7
1. Introduction	12
1.1. Intended Audience	12
1.2. Lightweight Persistence	12
2. Why JPA?	13
3. EJB Persistence Architecture	15
3.1. EJB Exceptions	16
4. Entity	18
4.1. Restrictions on Persistent Classes	19
4.1.1. Default or No-Arg Constructor	19
4.1.2. Final	19
4.1.3. Identity Fields	19
4.1.4. Version Field	19
4.1.5. Inheritance	20
4.1.6. Persistent Fields	20
4.1.7. Conclusions	21
4.2. Entity Identity	21
4.2.1. Identity Class	22
4.2.1.1. Identity Hierarchies	23
4.3. Lifecycle Callbacks	24
4.3.1. Callback Methods	24
4.3.2. Using Callback Methods	25
4.3.3. Using Entity Listeners	26
4.3.4. Entity Listeners Hierarchy	26
4.4. Conclusions	27
5. Metadata	28
5.1. Class Metadata	29
5.1.1. Entity	29
5.1.2. Id Class	30
5.1.3. Mapped Superclass	30
5.1.4. Embeddable	30
5.1.5. EntityListeners	31
5.1.6. Example	31
5.2. Field and Property Metadata	33
5.2.1. Transient	34
5.2.2. Id	34

5.2.3. Generated Value	34
5.2.4. Embedded Id	35
5.2.5. Version	35
5.2.6. Basic	35
5.2.6.1. Fetch Type	36
5.2.7. Embedded	36
5.2.8. Many-To-One	36
5.2.8.1. Cascade Type	37
5.2.9. One-To-Many	38
5.2.9.1. Bidirectional Relations	39
5.2.10. One-To-One	39
5.2.11. Many-To-Many	40
5.2.12. Order By	41
5.2.13. Map Key	41
5.2.14. Persistent Field Defaults	41
5.3. XML Schema	42
5.4. Conclusion	58
6. Persistence	63
6.1. persistence.xml	63
6.2. Non-EE Use	65
7. EntityManagerFactory	66
7.1. Obtaining an EntityManagerFactory	66
7.2. Obtaining EntityManagers	66
7.3. Persistence Context	67
7.3.1. Transaction Persistence Context	67
7.3.2. Extended Persistence Context	68
7.4. Closing the EntityManagerFactory	69
8. EntityManager	70
8.1. Transaction Association	70
8.2. Entity Lifecycle Management	71
8.3. Lifecycle Examples	74
8.4. Entity Identity Management	76
8.5. Cache Management	76
8.6. Query Factory	77
8.7. Closing	78
9. Transaction	79
9.1. Transaction Types	79
9.2. The EntityTransaction Interface	80
10. JPA Query	82
10.1. JPQL API	82
10.1.1. Query Basics	82
10.1.2. Relation Traversal	85
10.1.3. Fetch Joins	86
10.1.4. JPQL Functions	86
10.1.5. Polymorphic Queries	88
10.1.6. Query Parameters	88
10.1.7. Ordering	89
10.1.8. Aggregates	89
10.1.9. Named Queries	90
10.1.10. Delete By Query	90
10.1.11. Update By Query	91
10.2. JPQL Language Reference	91
10.2.1. JPQL Statement Types	92
10.2.1.1. JPQL Select Statement	92
10.2.1.2. JPQL Update and Delete Statements	92
10.2.2. JPQL Abstract Schema Types and Query Domains	92
10.2.2.1. JPQL Entity Naming	93
10.2.2.2. JPQL Schema Example	93
10.2.3. JPQL FROM Clause and Navigational Declarations	93

10.2.3.1. JPQL FROM Identifiers	94
10.2.3.2. JPQL Identification Variables	96
10.2.3.3. JPQL Range Declarations	96
10.2.3.4. JPQL Path Expressions	97
10.2.3.5. JPQL Joins	98
10.2.3.5.1. JPQL Inner Joins (Relationship Joins)	98
10.2.3.5.2. JPQL Outer Joins	98
10.2.3.5.3. JPQL Fetch Joins	99
10.2.3.6. JPQL Collection Member Declarations	99
10.2.3.7. JPQL Polymorphism	100
10.2.4. JPQL WHERE Clause	100
10.2.5. JPQL Conditional Expressions	100
10.2.5.1. JPQL Literals	100
10.2.5.2. JPQL Identification Variables	101
10.2.5.3. JPQL Path Expressions	101
10.2.5.4. JPQL Input Parameters	101
10.2.5.4.1. JPQL Positional Parameters	101
10.2.5.4.2. JPQL Named Parameters	101
10.2.5.5. JPQL Conditional Expression Composition	101
10.2.5.6. JPQL Operators and Operator Precedence	102
10.2.5.7. JPQL Between Expressions	102
10.2.5.8. JPQL In Expressions	103
10.2.5.9. JPQL Like Expressions	104
10.2.5.10. JPQL Null Comparison Expressions	104
10.2.5.11. JPQL Empty Collection Comparison Expressions	104
10.2.5.12. JPQL Collection Member Expressions	105
10.2.5.13. JPQL Exists Expressions	105
10.2.5.14. JPQL All or Any Expressions	105
10.2.5.15. JPQL Subqueries	106
10.2.5.16. JPQL Functional Expressions	106
10.2.5.16.1. JPQL String Functions	107
10.2.5.16.2. JPQL Arithmetic Functions	107
10.2.5.16.3. JPQL Datetime Functions	107
10.2.6. JPQL GROUP BY, HAVING	107
10.2.7. JPQL SELECT Clause	108
10.2.7.1. JPQL Result Type of the SELECT Clause	109
10.2.7.2. JPQL Constructor Expressions	109
10.2.7.3. JPQL Null Values in the Query Result	109
10.2.7.4. JPQL Aggregate Functions	109
10.2.7.4.1. JPQL Aggregate Examples	110
10.2.8. JPQL ORDER BY Clause	110
10.2.9. JPQL Bulk Update and Delete	111
10.2.10. JPQL Null Values	111
10.2.11. JPQL Equality and Comparison Semantics	112
10.2.12. JPQL BNF	112
11. SQL Queries	116
11.1. Creating SQL Queries	116
11.2. Retrieving Persistent Objects with SQL	116
12. Mapping Metadata	118
12.1. Table	119
12.2. Unique Constraints	122
12.3. Column	123
12.4. Identity Mapping	123
12.5. Generators	126
12.5.1. Sequence Generator	126
12.5.2. TableGenerator	127
12.5.3. Example	128
12.6. Inheritance	129
12.6.1. Single Table	130

12.6.1.1. Advantages	131
12.6.1.2. Disadvantages	131
12.6.2. Joined	131
12.6.2.1. Advantages	133
12.6.2.2. Disadvantages	133
12.6.3. Table Per Class	134
12.6.3.1. Advantages	135
12.6.3.2. Disadvantages	135
12.6.4. Putting it All Together	135
12.7. Discriminator	138
12.8. Field Mapping	143
12.8.1. Basic Mapping	143
12.8.1.1. LOBs	143
12.8.1.2. Enumerated	143
12.8.1.3. Temporal Types	144
12.8.1.4. The Updated Mappings	144
12.8.2. Secondary Tables	149
12.8.3. Embedded Mapping	150
12.8.4. Direct Relations	153
12.8.5. Join Table	156
12.8.6. Bidirectional Mapping	158
12.8.7. Map Mapping	158
12.9. The Complete Mappings	160
13. Conclusion	168
3. Java Data Objects	169
1. Introduction	173
1.1. Intended Audience	173
1.2. Transparent Persistence	173
2. Why JDO?	174
3. JDO Architecture	176
3.1. JDO Exceptions	177
4. PersistenceCapable	179
4.1. Enhancer	179
4.2. Persistence-Capable vs. Persistence-Aware	180
4.3. Restrictions on Persistent Classes	180
4.3.1. Default or No-Arg Constructor	180
4.3.2. Inheritance	180
4.3.3. Persistent Fields	181
4.3.4. Conclusions	183
4.4. Lifecycle Callbacks	183
4.4.1. InstanceCallbacks	183
4.4.2. InstanceLifecycleListener	185
4.5. JDO Identity	186
4.5.1. Datastore Identity	187
4.5.2. Application Identity	187
4.5.2.1. Application Identity Hierarchies	190
4.5.3. Single Field Identity	190
4.6. Conclusions	192
5. Metadata	193
5.1. Persistence Metadata DTD	193
5.2. JDO, Package, and Extension Elements	194
5.3. Class Element	195
5.4. Field Element	196
5.5. Fetch Group Element	198
5.6. The Complete Document	199
5.7. Metadata Placement	200
6. JDOHelper	201
6.1. Persistence-Capable Operations	201
6.2. Lifecycle Operations	202

6.3. PersistenceManagerFactory Construction	205
7. PersistenceManagerFactory	207
7.1. Obtaining a PersistenceManagerFactory	207
7.2. PersistenceManagerFactory Properties	208
7.2.1. Connection Configuration	208
7.2.2. PersistenceManager and Transaction Defaults	209
7.3. Obtaining PersistenceManagers	211
7.4. Properties and Supported Options	211
7.5. DataStoreCache Access	213
7.6. Closing the PersistenceManagerFactory	213
8. PersistenceManager	214
8.1. User Object Association	215
8.2. Configuration Properties	215
8.3. Transaction Association	216
8.4. FetchPlan Association	216
8.5. Persistence-Capable Lifecycle Management	216
8.6. Lifecycle Examples	218
8.7. Detach and Attach Functionality	219
8.8. JDO Identity Management	221
8.9. Cache Management	222
8.10. Extent Factory	223
8.11. Query Factory	223
8.12. Sequence Factory	224
8.13. Connection Access	224
8.14. Closing	225
9. Transaction	226
9.1. Transaction Types	226
9.2. The JDO Transaction Interface	227
9.2.1. Transaction Properties	227
9.2.2. Transaction Demarcation	228
10. Extent	230
11. Query	232
11.1. Object Filtering	232
11.2. JDOQL	234
11.3. Advanced Object Filtering	236
11.4. Compiling and Executing Queries	239
11.5. Limits and Ordering	240
11.6. Projections	241
11.7. Aggregates	244
11.8. Result Class	246
11.8.1. JavaBean Result Class	246
11.8.2. Generic Result Class	248
11.9. Single-String JDOQL	248
11.10. Named Queries	250
11.10.1. Defining Named Queries	250
11.10.2. Executing Named Queries	252
11.11. Delete By Query	252
11.12. Conclusion	253
12. FetchPlan	254
12.1. Detachment Options	255
13. DataStoreCache	257
14. JDOR	259
15. Mapping Metadata	260
15.1. Mapping Metadata Placement	260
15.2. Mapping Metadata DTD	261
15.3. Sequences	263
15.4. Class Table	264
15.5. Datastore Identity	266
15.6. Column	268

15.7. Joins	269
15.7.1. Shortcuts	270
15.7.2. Self Joins	271
15.7.3. Target Fields	272
15.8. Inheritance	273
15.8.1. subclass-table	273
15.8.1.1. Advantages	274
15.8.1.2. Disadvantages	274
15.8.1.3. Additional Considerations	274
15.8.2. new-table	275
15.8.2.1. Joined	275
15.8.2.1.1. Advantages	277
15.8.2.1.2. Disadvantages	277
15.8.2.2. Table Per Class	277
15.8.2.2.1. Advantages	278
15.8.2.2.2. Disadvantages	278
15.8.3. superclass-table	279
15.8.3.1. Advantages	279
15.8.3.2. Disadvantages	279
15.8.4. Putting it All Together	279
15.9. Discriminator	281
15.9.1. class-name	281
15.9.2. value-map	282
15.9.3. none	282
15.9.4. Putting it All Together	282
15.10. Version	284
15.10.1. none	285
15.10.2. version-number	285
15.10.3. date-time	285
15.10.4. state-comparison	285
15.10.5. Putting it All Together	286
15.11. Field Mapping	287
15.11.1. Superclass Fields	288
15.11.2. Basic Mapping	288
15.11.2.1. CLOB	290
15.11.2.2. BLOB	291
15.11.3. Automatic Values	291
15.11.4. Secondary Tables	292
15.11.5. Direct Relations	294
15.11.5.1. Inverse Keys	295
15.11.5.2. Bidirectional Relations	296
15.11.6. Basic Collections	297
15.11.7. Association Table Collections	298
15.11.7.1. Bidirectional Relations	301
15.11.8. Inverse Key Collections	301
15.11.8.1. Bidirectional Relations	303
15.11.9. Maps	304
15.11.10. Embedded Objects	306
15.12. Foreign Keys	309
15.13. Indexes	311
15.14. Unique Constraints	312
15.15. The Complete Document	313
16. Sequence	318
17. SQL Queries	320
17.1. Creating SQL Queries	320
17.2. Retrieving Persistent Objects with SQL	321
17.3. SQL Projections	322
17.4. Named SQL Queries	324
17.5. Conclusion	324

18. Conclusion	325
4. Tutorials	326
1. JPA Tutorials	329
1.1. Kodo JPA Tutorials	329
1.1.1. Tutorial Requirements	329
1.2. Kodo JPA Tutorial	329
1.2.1. The Pet Shop	329
1.2.1.1. Included Files	329
1.2.1.2. Important Utilities	330
1.2.2. Getting Started	330
1.2.2.1. Configuring the Datastore	332
1.2.3. Inventory Maintenance	333
1.2.3.1. Persisting Objects	334
1.2.3.2. Deleting Objects	335
1.2.4. Inventory Growth	336
1.2.5. Behavioral Analysis	338
1.2.5.1. Complex Queries	341
1.2.6. Extra Features	342
1.3. J2EE Tutorial	342
1.3.1. Prerequisites for the Kodo J2EE Tutorial	342
1.3.2. J2EE Installation Types	342
1.3.3. Installing Kodo Into Pre-J2EE 5 Application Servers	343
1.3.4. Installing the J2EE Sample Application	343
1.3.4.1. Compiling and Building The Sample Application	343
1.3.4.2. Deploying Sample To JBoss	344
1.3.4.3. Deploying Sample To WebLogic 9	344
1.3.5. Using The Sample Application	344
1.3.6. Sample Architecture	344
1.3.7. Code Notes and J2EE Tips	345
2. JDO Tutorials	346
2.1. Kodo JDO Tutorials	346
2.1.1. Tutorial Requirements	346
2.2. Kodo JDO Tutorial	346
2.2.1. The Pet Shop	346
2.2.1.1. Included Files	346
2.2.1.2. Important Utilities	347
2.2.2. Getting Started	347
2.2.2.1. Configuring the Datastore	348
2.2.3. Inventory Maintenance	349
2.2.3.1. Persisting Objects	350
2.2.3.2. Deleting Objects	351
2.2.4. Inventory Growth	352
2.2.5. Behavioral Analysis	354
2.2.5.1. Complex Queries	357
2.2.6. Extra Features	358
2.3. Reverse Mapping Tool Tutorial	358
2.3.1. Magazine Shop	358
2.3.2. Setup	359
2.3.2.1. Tutorial Files	359
2.3.2.2. Important Utilities	359
2.3.3. Generating Persistent Classes	359
2.3.4. Using the Finder	361
2.4. J2EE Tutorial	362
2.4.1. Prerequisites for the Kodo J2EE Tutorial	362
2.4.2. J2EE Installation Types	363
2.4.3. Installing Kodo	363
2.4.4. Installing the J2EE Sample Application	363
2.4.4.1. Compiling and Building The Sample Application	363
2.4.4.2. Deploying Sample To JBoss	364

2.4.4.3. Deploying Sample To WebLogic 6.1 to 7.x	364
2.4.4.4. Deploying Sample To WebLogic 8.1	364
2.4.4.5. Deploying Sample To SunONE / Sun JES	365
2.4.4.6. Deploying Sample To JRun	365
2.4.4.7. Deploying Sample To WebSphere	365
2.4.4.8. Deploying Sample To Borland Enterprise Server 5.2	365
2.4.5. Using The Sample Application	365
2.4.6. Sample Architecture	365
2.4.7. Code Notes and J2EE Tips	366
5. Kodo Frequently Asked Questions	369
1. Kodo JPA/JDO Frequently Asked Questions	371
1.1. General	371
1.2. Database	373
1.3. Programming with Kodo	375
1.4. How do I ... ?	378
1.5. Common errors	379
1.6. Productivity tools	380
1.7. Performance	380
1.8. Scalability	381
1.9. Application servers	381
1.10. Locking	382
1.11. Transactions	382
6. Kodo JPA/JDO Reference Guide	384
1. Introduction	392
1.1. Intended Audience	392
2. Configuration	393
2.1. Introduction	393
2.2. License Configuration	393
2.3. Runtime Configuration	393
2.4. Command Line Configuration	394
2.4.1. Code Formatting	395
2.5. Plugin Configuration	395
2.6. JDO Standard Properties	397
2.6.1. javax.jdo.PersistenceManagerFactoryClass	397
2.7. Kodo Properties	397
2.7.1. kodo.AggregateListeners	397
2.7.2. kodo.AutoClear	397
2.7.3. kodo.AutoDetach	398
2.7.4. kodo.BrokerFactory	398
2.7.5. kodo.BrokerImpl	398
2.7.6. kodo.ClassResolver	398
2.7.7. kodo.Compatibility	398
2.7.8. kodo.ConnectionDriverName	399
2.7.9. kodo.Connection2DriverName	399
2.7.10. kodo.ConnectionFactory	399
2.7.11. kodo.ConnectionFactory2	399
2.7.12. kodo.ConnectionFactoryName	400
2.7.13. kodo.ConnectionFactory2Name	400
2.7.14. kodo.ConnectionFactoryMode	400
2.7.15. kodo.ConnectionFactoryProperties	400
2.7.16. kodo.ConnectionFactory2Properties	401
2.7.17. kodo.ConnectionPassword	401
2.7.18. kodo.Connection2Password	401
2.7.19. kodo.ConnectionProperties	401
2.7.20. kodo.Connection2Properties	401
2.7.21. kodo.ConnectionURL	402
2.7.22. kodo.Connection2URL	402
2.7.23. kodo.ConnectionUserName	402
2.7.24. kodo.Connection2UserName	402

2.7.25. kodo.ConnectionRetainMode	403
2.7.26. kodo.DataCache	403
2.7.27. kodo.DataCacheTimeout	403
2.7.28. kodo.DetachState	403
2.7.29. kodo.DynamicDataStructs	404
2.7.30. kodo.FetchBatchSize	404
2.7.31. kodo.FetchGroups	404
2.7.32. kodo.FilterListeners	404
2.7.33. kodo.FlushBeforeQueries	405
2.7.34. kodo.IgnoreChanges	405
2.7.35. kodo.InverseManager	405
2.7.36. kodo.LicenseKey	405
2.7.37. kodo.LockManager	406
2.7.38. kodo.LockTimeout	406
2.7.39. kodo.Log	406
2.7.40. kodo.ManagedRuntime	406
2.7.41. kodo.ManagementConfiguration	407
2.7.42. kodo.Mapping	407
2.7.43. kodo.MetaDataFactory	407
2.7.44. kodo.Multithreaded	407
2.7.45. kodo.Optimistic	408
2.7.46. kodo.OrphanedKeyAction	408
2.7.47. kodo.NontransactionalRead	408
2.7.48. kodo.NontransactionalWrite	408
2.7.49. kodo.PersistenceServer	408
2.7.50. kodo.ProxyManager	409
2.7.51. kodo.QueryCache	409
2.7.52. kodo.QueryCompilationCache	409
2.7.53. kodo.ReadLockLevel	409
2.7.54. kodo.RemoteCommitProvider	410
2.7.55. kodo.RestoreState	410
2.7.56. kodo.RetainState	410
2.7.57. kodo.RetryClassRegistration	410
2.7.58. kodo.SavepointManager	411
2.7.59. kodo.Sequence	411
2.7.60. kodo.TransactionMode	411
2.7.61. kodo.WriteLockLevel	411
2.8. Kodo JDBC Properties	412
2.8.1. kodo.jdbc.ConnectionDecorators	412
2.8.2. kodo.jdbc.DBDictionary	412
2.8.3. kodo.jdbc.EagerFetchMode	412
2.8.4. kodo.jdbc.FetchDirection	413
2.8.5. kodo.jdbc.JDBCListeners	413
2.8.6. kodo.jdbc.LRSSize	413
2.8.7. kodo.jdbc.MappingDefaults	413
2.8.8. kodo.jdbc.MappingFactory	414
2.8.9. kodo.jdbc.ResultSetType	414
2.8.10. kodo.jdbc.Schema	414
2.8.11. kodo.jdbc.SchemaFactory	414
2.8.12. kodo.jdbc.Schemas	415
2.8.13. kodo.jdbc.SubclassFetchMode	415
2.8.14. kodo.jdbc.SynchronizeMappings	415
2.8.15. kodo.jdbc.TransactionIsolation	415
2.8.16. kodo.jdbc.UpdateManager	416
3. Logging	417
3.1. Logging Channels	417
3.2. Kodo Logging	418
3.3. Disabling Logging	419
3.4. Log4J	420

3.5. Apache Commons Logging	420
3.5.1. JDK 1.4 java.util.logging	420
3.6. Custom Log	421
4. JDBC	423
4.1. Using the Kodo DataSource	423
4.2. Using a Third-Party DataSource	426
4.2.1. Managed and XA DataSources	426
4.3. Runtime Access to DataSource	427
4.4. Database Support	428
4.4.1. DBDictionary Properties	430
4.4.2. MySQLDictionary Properties	435
4.4.3. OracleDictionary Properties	435
4.5. Setting the Transaction Isolation	436
4.6. Setting the SQL Join Syntax	436
4.7. Accessing Multiple Databases	438
4.8. Configuring the Use of JDBC Connections	438
4.9. Statement Batching	440
4.10. Large Result Sets	440
4.11. Default Schema	443
4.12. Schema Reflection	443
4.12.1. Schemas List	443
4.12.2. Schema Factory	443
4.13. Schema Tool	444
4.14. XML Schema Format	447
4.15. The SQLLine Utility	449
5. Persistent Classes	452
5.1. Persistent Class List	452
5.2. Enhancement	452
5.2.1. Enhancing at Build Time	453
5.2.2. Enhancing JPA Entities on Deployment	454
5.2.3. Enhancing at Runtime	454
5.2.4. Serializing Enhanced Types	455
5.3. Object Identity	455
5.3.1. Datastore Identity Objects	456
5.3.2. Application Identity Tool	456
5.3.3. Autoassign / Identity Strategy Caveats	458
5.4. Managed Inverses	458
5.5. Persistent Fields	460
5.5.1. Restoring State	460
5.5.2. Typing and Ordering	460
5.5.3. Calendar Fields and TimeZones	461
5.5.4. Proxies	461
5.5.4.1. Smart Proxies	461
5.5.4.2. Large Result Set Proxies	461
5.5.4.3. Custom Proxies	463
5.5.5. Externalization	464
5.5.5.1. External Values	468
5.6. Fetch Groups	469
5.6.1. Custom Fetch Groups	469
5.6.2. Custom Fetch Group Configuration	470
5.6.3. Per-field Fetch Configuration	471
5.6.4. Implementation Notes	472
5.7. Eager Fetching	473
5.7.1. Configuring Eager Fetching	474
5.7.2. Eager Fetching Considerations and Limitations	475
5.8. Lock Groups	476
5.8.1. Lock Groups and Subclasses	477
5.8.2. Lock Group Mapping	479
6. Metadata	480

6.1. Generating Default JDO Metadata	480
6.2. Metadata Factory	480
6.3. Additional JPA Metadata	481
6.3.1. Datastore Identity	481
6.3.2. Surrogate Version	482
6.3.3. Persistent Field Values	482
6.3.4. Persistent Collection Fields	482
6.3.5. Persistent Map Fields	483
6.4. Metadata Extensions	483
6.4.1. Class Extensions	483
6.4.1.1. Fetch Groups	483
6.4.1.2. Data Cache	483
6.4.1.3. Detached State	484
6.4.1.4. Lock Groups	484
6.4.1.5. Auditable	485
6.4.2. Field Extensions	485
6.4.2.1. Dependent	485
6.4.2.2. LRS	485
6.4.2.3. Order-By	485
6.4.2.4. Inverse-Logical	486
6.4.2.5. Lock Group	486
6.4.2.6. Read-Only	486
6.4.2.7. Type	486
6.4.2.8. Externalizer	487
6.4.2.9. Factory	487
6.4.2.10. External Values	488
6.4.3. Example	488
7. Mapping	489
7.1. Forward Mapping	489
7.1.1. Using the Mapping Tool	491
7.1.2. Generating DDL SQL	493
7.1.3. JDO Forward Mapping Hints	494
7.1.4. Runtime Forward Mapping	495
7.2. Reverse Mapping	495
7.2.1. Customizing Reverse Mapping	498
7.3. Meet-in-the-Middle Mapping	499
7.4. Mapping Defaults	500
7.5. Mapping Factory	503
7.5.1. Importing and Exporting Mapping Data	505
7.6. Non-Standard Joins	506
7.7. Additional JPA Mappings	508
7.7.1. Datastore Identity Mapping	508
7.7.2. Surrogate Version Mapping	509
7.7.3. Multi-Column Mappings	510
7.7.4. Join Column Attribute Targets	510
7.7.5. Embedded Mapping	510
7.7.6. Collections	512
7.7.6.1. Container Table	512
7.7.6.2. Element Columns	512
7.7.6.3. Element Join Columns	512
7.7.6.4. Element Embedded Mapping	512
7.7.6.5. Order Column	513
7.7.6.6. Examples	513
7.7.7. One-Sided One-Many Mapping	514
7.7.8. Maps	515
7.7.8.1. Key Columns	515
7.7.8.2. Key Join Columns	516
7.7.8.3. Key Embedded Mapping	516
7.7.8.4. Examples	516

7.7.9. Indexes and Constraints	517
7.7.9.1. Indexes	517
7.7.9.2. Foreign Keys	517
7.7.9.3. Unique Constraints	518
7.7.9.4. Examples	518
7.8. Mapping Limitations	519
7.8.1. Table Per Class	519
7.9. Mapping Extensions	519
7.9.1. Class Extensions	520
7.9.1.1. Subclass Fetch Mode	520
7.9.1.2. Strategy	520
7.9.1.3. Discriminator Strategy	520
7.9.1.4. Version Strategy	520
7.9.2. Field Extensions	520
7.9.2.1. Eager Fetch Mode	520
7.9.2.2. Nonpolymorphic	520
7.9.2.3. Class Criteria	521
7.9.2.4. Strategy	522
7.9.3. Column Extensions	522
7.9.3.1. insertable	522
7.9.3.2. updatable	522
7.9.3.3. lock-group	522
7.10. Custom Mappings	522
7.10.1. Custom Class Mapping	523
7.10.2. Custom Discriminator and Version Strategies	523
7.10.3. Custom Field Mapping	523
7.10.3.1. Value Handlers	523
7.10.3.2. Field Strategies	523
7.10.3.3. Configuration	524
7.11. Orphaned Keys	525
8. Deployment	527
8.1. Factory Deployment	527
8.1.1. Standalone Deployment	527
8.1.2. EntityManager Injection	527
8.1.3. Kodo JPA JCA Deployment	527
8.1.3.1. WebLogic 9	527
8.1.3.2. JBoss 4.x	528
8.1.4. Kodo JDO JCA Deployment	528
8.1.4.1. WebLogic 6.1 to 7.x	528
8.1.4.2. WebLogic 8.1	529
8.1.4.3. WebLogic 9	530
8.1.4.4. JBoss 3.0	530
8.1.4.5. JBoss 3.2	530
8.1.4.6. JBoss 4.x	531
8.1.4.7. WebSphere 5	531
8.1.4.8. SunONE Application Server 7 / Sun Java Enterprise Server 8-8.1	531
8.1.4.9. Macromedia JRun 4	532
8.1.4.10. Borland Enterprise Server 5.2 - 6.0	532
8.1.5. Non-JCA Application Server Deployment	533
8.2. Integrating with the Transaction Manager	535
8.3. XA Transactions	536
8.3.1. Using Kodo with XA Transactions	536
9. Runtime Extensions	538
9.1. Architecture	538
9.1.1. Broker Customization	539
9.2. JPA Extensions	540
9.2.1. KodoEntityManagerFactory	540
9.2.2. KodoEntityManager	540
9.2.3. KodoQuery	540

9.2.4. Extent	540
9.2.5. StoreCache	541
9.2.6. QueryResultCache	541
9.2.7. FetchPlan	541
9.2.8. KodoPersistence	541
9.3. JDO API Extensions	541
9.3.1. KodoPersistenceManagerFactory	541
9.3.2. KodoPersistenceManager	541
9.3.3. KodoQuery	542
9.3.4. KodoExtent	542
9.3.5. KodoDataStoreCache	542
9.3.6. QueryResultCache	542
9.3.7. KodoFetchPlan	542
9.3.8. KodoJDOHelper	542
9.4. Object Locking	542
9.4.1. Configuring Default Locking	542
9.4.2. Configuring Lock Levels at Runtime	543
9.4.3. Object Locking APIs	544
9.4.4. Lock Manager	546
9.4.5. Rules for Locking Behavior	547
9.4.6. Known Issues and Limitations	547
9.5. Savepoints	548
9.5.1. Using Savepoints	548
9.5.2. Configuring Savepoints	549
9.6. Query Language Extensions	550
9.6.1. Filter Extensions	550
9.6.1.1. Included Filter Extensions	551
9.6.1.2. Developing Custom Filter Extensions	552
9.6.1.3. Configuring Filter Extensions	552
9.6.2. Aggregate Extensions	552
9.6.2.1. Configuring Query Aggregates	552
9.6.3. JDOQL Non-Distinct Results	552
9.6.4. JDOQL Subqueries	553
9.6.4.1. Subquery Parameters, Variables, and Imports	554
9.6.5. MethodQL	554
9.7. Generators	555
9.7.1. Runtime Access	558
9.8. Transaction Events	559
9.9. Non-Relational Stores	559
10. Caching	560
10.1. Data Cache	560
10.1.1. Data Cache Configuration	561
10.1.2. Data Cache Usage	563
10.1.3. Query Cache	566
10.1.4. The Concurrent Data Cache	571
10.1.5. Third-Party Integrations	572
10.1.5.1. Tangosol Integration	572
10.1.5.2. GemStone GemFire Integration	573
10.1.6. Cache Extension	574
10.1.7. Important Notes	575
10.1.8. Known Issues and Limitations	575
10.2. Query Compilation Cache	576
11. Remote and Offline Operation	577
11.1. Detach and Attach	577
11.1.1. Detach Behavior	577
11.1.2. Attach Behavior	577
11.1.3. Defining the Detached Object Graph	578
11.1.3.1. Detached State Field	580
11.1.4. Automatic Detachment	580

11.2. Remote Managers	581
11.2.1. Standalone Persistence Server	582
11.2.2. HTTP Persistence Server	583
11.2.3. Client Managers	583
11.2.4. Data Compression and Filtering	586
11.2.5. Remote Persistence Deployment	587
11.2.6. Remote Transfer Listeners	587
11.3. Remote Event Notification Framework	587
11.3.1. Remote Commit Provider Configuration	587
11.3.1.1. JMS	587
11.3.1.2. TCP	588
11.3.1.3. Common Properties	589
11.3.2. Customization	590
12. Management and Monitoring	591
12.1. Configuration	591
12.1.1. Optional Parameters in Management Group	593
12.1.2. Optional Parameters in Remote Group	594
12.1.3. Optional Parameters in JSR 160 Group	594
12.1.4. Optional Parameters in WebLogic 8.1 Group	595
12.1.5. Configuring Logging for Management / Monitoring	595
12.2. Kodo Management Console	595
12.2.1. Remote Connection	595
12.2.1.1. Connecting to Kodo under WebLogic 8.1	596
12.2.1.2. Connecting to Kodo under JBoss 3.2	596
12.2.1.3. Connecting to Kodo under JBoss 4	597
12.2.2. Using the Kodo Management Console	597
12.2.2.1. JMX Explorer	598
12.2.2.1.1. Executing Operations	598
12.2.2.1.2. Listening to Notifications	598
12.2.2.2. MBean Panel	599
12.2.2.2.1. Notifications / Statistics	599
12.2.2.2.2. Setting Attributes	599
12.3. Accessing the MBeanServer from Code	599
12.4. MBeans	600
12.4.1. Log MBean	600
12.4.2. Kodo Pooling DataSource MBean	600
12.4.3. Prepared Statement Cache MBean	600
12.4.4. Query Cache MBean	600
12.4.5. Data Cache MBean	600
12.4.6. TimeWatch MBean	600
12.4.7. Runtime MBean	601
12.4.8. Profiling MBean	601
13. Profiling	603
13.1. Profiling in an Embedded GUI	603
13.2. Dumping Profiling Data to Disk from a Batch Process	605
13.3. Controlling How the Profiler Obtains Context Information	606
14. Third Party Integration	607
14.1. Apache Ant	607
14.1.1. Common Ant Configuration Options	607
14.1.2. Enhancer Ant Task	609
14.1.3. Application Identity Tool Ant Task	609
14.1.4. Mapping Tool Ant Task	610
14.1.5. Reverse Mapping Tool Ant Task	610
14.1.6. Schema Tool Ant Task	611
14.2. Maven	611
15. Optimization Guidelines	612
7. Kodo JPA/JDO Samples	616
1. Kodo Sample Code	618
1.1. JDO - JPA Persistence Interoperability	618

1.2. JPA	619
1.2.1. Sample Human Resources Model	619
1.2.2. JMX Management	619
1.3. JDO	620
1.3.1. Using Application Identity	620
1.3.2. Customizing Logging	620
1.3.3. Custom Sequence Factory	620
1.3.4. Using Externalization to Persist Second Class Objects	620
1.3.5. Custom Mappings	621
1.3.6. Custom Proxies	621
1.3.7. Example of full-text searching in JDO	621
1.3.8. Horizontal Mappings	622
1.3.9. Sample Human Resources Model	622
1.3.10. Sample School Schedule Model	622
1.3.11. JMX Management	623
1.3.12. XML Store Manager	624
1.3.13. Using JDO with Java Server Pages (jsp)	625
1.3.14. JDO Enterprise Java Beans 2.x Facade	625
1. JPA Resources	626
2. JDO Resources	627
3. Kodo Editions	628
3.1. Standard Edition	628
3.2. Performance Pack	628
3.3. Enterprise Edition	628
4. Supported Databases	629
4.1. Apache Derby	629
4.2. Borland Interbase	629
4.2.1. Known issues with Interbase	630
4.3. JDataStore	630
4.4. IBM DB2	630
4.4.1. Known issues with DB2	630
4.5. Empress	630
4.5.1. Known issues with Empress	631
4.6. Hypersonic	631
4.6.1. Known issues with Hypersonic	631
4.7. Firebird	631
4.7.1. Known issues with Firebird	632
4.8. Informix	632
4.8.1. Known issues with Informix	632
4.9. InterSystems Cache	632
4.9.1. Known issues with InterSystems Cache	632
4.10. Microsoft Access	632
4.10.1. Known issues with Microsoft Access	633
4.11. Microsoft SQL Server	633
4.11.1. Known issues with SQL Server	633
4.12. Microsoft FoxPro	633
4.12.1. Known issues with Microsoft FoxPro	634
4.13. MySQL	634
4.13.1. Known issues with MySQL	634
4.14. Oracle	634
4.14.1. Using Query Hints with Oracle	635
4.14.2. Known issues with Oracle	635
4.15. Pointbase	636
4.15.1. Known issues with Pointbase	636
4.16. PostgreSQL	636
4.16.1. Known issues with PostgreSQL	636
4.17. Sybase Adaptive Server	636
4.17.1. Known issues with Sybase	637
5. Common Database Errors	638

6. Upgrading Kodo	648
6.1. Compatibility Configuration	648
6.2. Migrating from Kodo 3.0 to Kodo 3.1	649
6.3. Migrating from Kodo 3.1 to Kodo 3.2	649
6.4. Migrating from Kodo 3.x to Kodo 4.0	649
6.4.1. API Compatibility	649
6.4.2. Metadata and Mappings	650
6.4.3. Configuration	650
7. Development and Runtime Libraries	651
8. Release Notes	652
Index	695

List of Tables

2.1. Persistence Mechanisms	13
2.1. Persistence Mechanisms	174
6.1. JDOHelper Lifecycle Methods	204
15.1. Default Value Inserts	269
4.1. Validation SQL Defaults	424
4.2. Kodo Automatic Flush Behavior	439
5.1. Externalizer Options	464
5.2. Factory Options	465
10.1. Data access methods	560
10.2. Pre-defined aliases	576
15.1. Optimization Guidelines	612
4.1. Supported Databases and JDBC Drivers	629
5.1. Known Database Error Codes	638

List of Examples

3.1. Interaction of Interfaces Outside Container	15
3.2. Interaction of Interfaces Inside Container	16
4.1. Persistent Class	18
4.2. Identity Class	22
5.1. Class Metadata	31
5.2. Complete Metadata	58
6.1. persistence.xml	64
6.2. Obtaining an EntityManagerFactory	65
7.1. Behavior of Transaction Persistence Context	68
7.2. Behavior of Extended Persistence Context	68
8.1. Persisting Objects	74
8.2. Updating Objects	74
8.3. Removing Objects	75
8.4. Detaching and Merging	75
9.1. Grouping Operations with Transactions	81
10.1. Delete by Query	91
10.2. Update by Query	91
11.1. Creating a SQL Query	116
11.2. Retrieving Persistent Objects	117
11.3. SQL Query Parameters	117
12.1. Mapping Classes	120
12.2. Defining a Unique Constraint	122
12.3. Identity Mapping	124
12.4. Generator Mapping	128
12.5. Single Table Mapping	130
12.6. Joined Subclass Tables	132
12.7. Table Per Class Mapping	134
12.8. Inheritance Mapping	136
12.9. Discriminator Mapping	140
12.10. Basic Field Mapping	145
12.11. Secondary Table Field Mapping	150
12.12. Embedded Field Mapping	151
12.13. Mapping Mapped Superclass Field	152
12.14. Direct Relation Field Mapping	154
12.15. Join Table Mapping	157
12.16. Join Table Map Mapping	159
12.17. Full Entity Mappings	161
3.1. Interaction of JDO Interfaces	177
4.1. PersistenceCapable Class	179
4.2. Accessing Mutable Persistent Fields	182
4.3. Using Callback Interfaces	184
4.4. JDO Identity Objects	186
4.5. Application Identity Class	188
5.1. Basic Structure of Metadata Documents	194
5.2. Metadata Class Listings	196
5.3. Full Metadata Document	199
6.1. Obtaining a PersistenceManagerFactory from Properties	205
6.2. Obtaining a PersistenceManagerFactory from a Resource	206
8.1. Persisting Objects	218
8.2. Updating Objects	218
8.3. Deleting Objects	219
8.4. Detaching and Attaching	220
8.5. Using the PersistenceManager's Connection	224
9.1. Grouping Operations with Transactions	229

10.1. Iterating an Extent	231
11.1. Filtering	233
11.2. Relation Traversal and Mathematical Operations	235
11.3. Precedence, Logical Operators, and String Functions	235
11.4. Collections	235
11.5. Static Methods	236
11.6. Imports and Declared Parameters	236
11.7. Implicit Parameters	237
11.8. Query By Example	238
11.9. Variables	238
11.10. Unconstrained Variables	238
11.11. Unique	241
11.12. Result Range and Ordering	241
11.13. Projection	242
11.14. Projection Field Traversal	242
11.15. Projection Variables	243
11.16. Distinct Projection	243
11.17. Count	244
11.18. Min, Max, Avg	245
11.19. Grouping	245
11.20. Populating a JavaBean	247
11.21. Result Aliases	247
11.22. Taking Advantage of the Default Result String	248
11.23. Populating a Map	248
11.24. Query Metadata Document	251
11.25. Persistence Metadata Document	251
11.26. Executing Named Queries	252
11.27. Delete By Query	253
12.1. Using the FetchPlan	255
12.2. Using Detachment Options	255
13.1. Using the DataStoreCache	258
15.1. Named Sequences	264
15.2. Mapping Classes	265
15.3. Datastore Identity Mapping	267
15.4. subclass-table Mapping	273
15.5. Joined Subclass Tables	276
15.6. Table Per Class Mapping	278
15.7. superclass-table Mapping	279
15.8. Inheritance Mapping	280
15.9. Discriminator Mapping	283
15.10. Version Mapping	286
15.11. Basic Field Mapping	289
15.12. CLOB Field Mapping	290
15.13. Byte Array Field Mapping	291
15.14. Serialized Field Mapping	291
15.15. Automatic Field Values	292
15.16. Secondary Table Field Mapping	293
15.17. Direct Relation Field Mapping	294
15.18. Inverse Key Relation Field Mapping	296
15.19. Mapping a Bidirectional Relation	296
15.20. Basic Collection Field Mapping	298
15.21. Association Table Collection Field Mapping	299
15.22. Mapping Bidirectional Association Table Relations	301
15.23. Inverse Key Collection Field Mapping	302
15.24. Direct Relation / Inverse Key Collection Pair	303
15.25. Map Field Mapping	304
15.26. Derived Key Mapping	305
15.27. Inverse Foreign Key Map Mapping	306
15.28. Embedded Field Mapping	307

15.29. Embedded Collection Elements	308
15.30. Using the delete-action Attribute	310
15.31. Using Contextual Foreign Key Elements	310
15.32. Using the indexed Attribute	311
15.33. Using Contextual Index Elements	312
15.34. Using the unique Attribute	312
15.35. Using Contextual Unique Elements	313
15.36. Full Mapping Document	314
15.37. Integrated JDO Metadata Document	316
16.1. Using Sequences	319
17.1. Creating a SQL Query	320
17.2. Retrieving Persistent Objects	321
17.3. SQL Query Parameters	322
17.4. SQL Query Parameter Map	322
17.5. Column Projection	323
17.6. Result Class	323
17.7. Named SQL Queries	324
1.1. Issuing a query against a Date field	378
2.1. Code Formatting with the Reverse Mapping Tool	395
3.1. Standard Kodo Log Configuration	418
3.2. Standard Kodo Log Configuration + All SQL Statements	419
3.3. Logging to a File	419
3.4. Standard Log4J Logging	420
3.5. JDK 1.4 Log Properties	421
3.6. Custom Logging Class	421
4.1. Properties for the Kodo DataSource	425
4.2. Properties File for a Third-Party DataSource	426
4.3. Managed DataSource Configuration	427
4.4. Using the EntityManager's Connection	427
4.5. Using the EntityManagerFactory's DataSource	428
4.6. Using the PersistenceManagerFactory DataSource	428
4.7. Specifying a DBDictionary	429
4.8. Specifying a Transaction Isolation	436
4.9. Specifying the Join Syntax Default	437
4.10. Specifying the Join Syntax at Runtime	437
4.11. Specifying Connection Usage Defaults	439
4.12. Specifying Connection Usage at Runtime	439
4.13. Configuring SQL Batching	440
4.14. Specifying Result Set Defaults	442
4.15. Specifying Result Set Behavior at Runtime	442
4.16. Schema Creation	446
4.17. SQL Scripting	446
4.18. Schema Drop	446
4.19. Schema Reflection	447
4.20. Basic Schema	448
4.21. Full Schema	448
4.22. Connecting to the Database	449
4.23. Connecting to the Database via Properties	449
4.24. Listing Commands	449
4.25. Examining the Tutorial Schema	450
4.26. Issuing SQL Against the Database	450
4.27. Disconnecting from the Database	451
5.1. Using the Kodo Enhancer	453
5.2. Using the Kodo Agent for Runtime Enhancement	455
5.3. Passing Options to the Kodo Agent	455
5.4. JPA Datastore Identity Metadata	456
5.5. Using the Application Identity Tool	457
5.6. Specifying Logical Inverses	458
5.7. Enabling Managed Inverses	459

5.8. Log Inconsistencies	460
5.9. Using Initial Field Values	460
5.10. Using a Large Result Set Iterator	462
5.11. Marking a Large Result Set Field	463
5.12. Configuring the Proxy Manager	463
5.13. Using Externalization	466
5.14. Querying Externalization Fields	467
5.15. Using External Values	468
5.16. Custom Fetch Group Metadata	470
5.17. Using the FetchPlan	471
5.18. Adding an Eager Field	472
5.19. Setting the Default Eager Fetch Mode	474
5.20. Setting the Eager Fetch Mode at Runtime	475
5.21. Lock Group Metadata	476
5.22. Lock Group Metadata	477
5.23. Mapping Lock Groups	479
6.1. Generating Metadata with the Mapping Tool	480
6.2. Setting a Standard Metadata Factory	481
6.3. Setting a Custom Metadata Factory	481
6.4. Kodo Metadata Extensions	488
7.1. Using the Mapping Tool	489
7.2. Creating the Relational Schema from Mappings	491
7.3. Refreshing Mappings and the Relational Schema	492
7.4. Adding Mappings and the Relational Schema	492
7.5. Dropping Mappings	492
7.6. Dropping Mappings and Association Schema	492
7.7. Create DDL for Current Mappings	493
7.8. Create DDL to Update Database for Current Mappings	493
7.9. Refresh JDO Mappings and Create DDL	493
7.10. Refresh JDO Mappings and Create DDL to Update Database	493
7.11. Partial Mapping	494
7.12. Configuring Runtime Forward Mapping	495
7.13. Reflection with the Schema Tool	496
7.14. Using the Reverse Mapping Tool	496
7.15. Customizing Reverse Mapping with Properties	499
7.16. Validating Mappings	500
7.17. Creating the Relational Schema from Mappings	500
7.18. Configuring Mapping Defaults	502
7.19. Standard JPA Configuration	504
7.20. Standard JDO Configuration	504
7.21. Recording Constraint Names	504
7.22. JDO ORM File Configuration	505
7.23. Storing JDO Mappings in a Table	505
7.24. JPA Metadata, JDO Mapping Files	505
7.25. Modifying Difficult-to-Access Mapping Data	505
7.26. Switching Mapping Factories	506
7.27. Datastore Identity Mapping	509
7.28. Overriding Complex Mappings	511
7.29. String List Mapping	513
7.30. Embedded Element Mapping	514
7.31. One-Sided One-Many Mapping	515
7.32. String Key, Entity Value Map Mapping	516
7.33. Constraint Mapping	518
7.34. Custom Mappings via Extensions	524
7.35. Custom Logging Orphaned Keys	525
8.1. Binding a Factory into JNDI via a WebLogic Startup Class	533
8.2. Looking up the Factory in JNDI	534
8.3. Configuring Transaction Manager Integration	536
9.1. Switching APIs within a Persistence Context	539

9.2. Evict from Data Cache	539
9.3. Using a JPA Extent	540
9.4. Setting Default Lock Levels	543
9.5. Setting Runtime Lock Levels	543
9.6. Locking APIs	545
9.7. Disabling Locking	547
9.8. Using Savepoints	549
9.9. Basic Filter Extension	550
9.10. Chaining Filter Extensions	550
9.11. Nondistinct JDOQL	553
9.12. Comparison to Subquery	553
9.13. Correlated Subquery	553
9.14. Subquery Contains	554
9.15. Subquery Empty	554
9.16. Named Seq Sequence	557
9.17. System Sequence Configuration	558
10.1. Single-JVM Data Cache	561
10.2. Data Cache Size	561
10.3. Data Cache Timeout	562
10.4. Named Data Cache Specification	563
10.5. Named Data Cache Configuration	563
10.6. Accessing the StoreCache	564
10.7. StoreCache Usage	565
10.8. Accessing a Named Cache	565
10.9. Automatic Data Cache Eviction	565
10.10. Accessing the QueryResultCache	566
10.11. Query Cache Size	567
10.12. Disabling the Query Cache	567
10.13. Evicting Queries	569
10.14. Pinning, and Unpinning Query Results	570
10.15. Disabling and Enabling Query Caching	570
10.16. Concurrent Data Cache Configuration	572
10.17. Tangosol Cache Configuration	572
10.18. Tangosol Query Cache Configuration	573
10.19. GemFire Cache Configuration	574
10.20. Query Replaces Extent	575
11.1. Configuring Detached State	579
11.2. Configuring a Standalone Persistence Server	582
11.3. Starting a Standalone Persistence Server	583
11.4. Client Configuration	585
11.5. HTTP Client Configuration	585
11.6. Enabling Compression with the TCP Transport	586
11.7. Enabling Compression with the HTTP Transport	586
11.8. JMS Remote Commit Provider Configuration	588
11.9. TCP Remote Commit Provider Configuration	589
11.10. Transmitting Persisted Object Ids	589
12.1. Accessing the MBeanServer	599
14.1. Using the <config> Ant Tag	607
14.2. Using the Properties Attribute of the <config> Tag	608
14.3. Using the PropertiesFile Attribute of the <config> Tag	608
14.4. Using the <classpath> Ant Tag	608
14.5. Using the <codeformat> Ant Tag	608
14.6. Invoking the Enhancer from Ant	609
14.7. Invoking the Application Identity Tool from Ant	609
14.8. Invoking the Mapping Tool from Ant	610
14.9. Invoking the Reverse Mapping Tool from Ant	610
14.10. Invoking the Schema Tool from Ant	611
4.1. Example properties for Derby	629
4.2. Example properties for Interbase	629

4.3. Example properties for JDataStore	630
4.4. Example properties for IBM DB2	630
4.5. Example properties for Empress	631
4.6. Example properties for Hypersonic	631
4.7. Example properties for Firebird	631
4.8. Example properties for Informix Dynamic Server	632
4.9. Example properties for InterSystems Cache	632
4.10. Example properties for Microsoft Access	633
4.11. Example properties for Microsoft SQLServer	633
4.12. Example properties for Microsoft FoxPro	633
4.13. Example properties for MySQL	634
4.14. Example properties for Oracle	634
4.15. Using Oracle Hints	635
4.16. Example properties for Pointbase	636
4.17. Example properties for PostgreSQL	636
4.18. Example properties for Sybase	636

Part 1. Introduction

Table of Contents

1. Kodo JPA/JDO	3
1.1. About This Document	3
1.2. Sales Inquiries	3
2. Kodo Installation	4
2.1. Overview	4
2.2. Installation	4
2.2.1. Updates	4
2.2.2. Key Files in the Download	4
2.2.3. Quick Start	4
2.2.4. Upgrading from Previous Kodo Versions	4
2.2.5. Requirements	5
2.2.6. Terminology	5
2.2.7. Windows Installation	5
2.2.8. POSIX (Linux, Solaris, Mac OS X, Windows with cygwin, etc.) Installation	5
2.2.9. Common Installation Problems	6
2.2.10. Resources	6

Chapter 1. Kodo JPA/JDO

Kodo JPA/JDO is BEA System's implementation of Sun's Java Persistence API (JPA) specification and Java Data Objects (JDO) specification for the transparent persistence of Java objects. This document provides an overview of these persistence standards and technical details on the use of Kodo JPA/JDO.

To quickly get started with JPA, you may want to begin at [Section 1.2, “Kodo JPA Tutorial” \[329\]](#) If you would prefer to start with an introduction to the concepts of JPA, begin with [Chapter 1, *Introduction* \[12\]](#)

To get familiar with JDO, you may want to start with [Section 2.2, “Kodo JDO Tutorial” \[346\]](#) If you are interested in reading about the background and concepts involved in JDO, start at [Chapter 1, *Introduction* \[173\]](#)

1.1. About This Document

This document is intended for Kodo users. It is divided into several parts:

- The **JPA Overview** describes the fundamentals of JPA.
- The **JDO Overview** describes the fundamentals of JDO.
- In the **Kodo JPA/JDO Tutorials** you will develop simple persistent applications using Kodo. Through the tutorials' hands-on approach, you will become comfortable with the core tools and development processes under Kodo JPA/JDO.
- The **Kodo JPA/JDO Reference Guide** contains detailed documentation on all aspects of Kodo JPA/JDO. Browse through this guide to familiarize yourself with the many advanced features and customization opportunities Kodo provides. Later, you can use the guide when you need details on a specific aspect of Kodo JPA/JDO.

1.2. Sales Inquiries

<sales@solarmetric.com>

+1 202-595-2064 x2

Chapter 2. Kodo Installation

2.1. Overview

This download includes the commercially available release of Kodo version 4.0.0. Kodo is an implementation of the Java Persistence API and Java Data Objects specifications for relational databases. A copy of each specification is provided in the documentation.

You just downloaded one of the following:

- `kodo-4.0.0.tar.gz` - Kodo version 4.0.0 for Unix or Mac OS X.
- `kodo-4.0.0.zip` - Kodo version 4.0.0 for Microsoft Windows XP, 2000, NT, ME, 98, and 95.

2.2. Installation

2.2.1. Updates

Please check the web site to download the latest version of Kodo.

2.2.2. Key Files in the Download

1. `README.txt` - The README file.
2. `lib/hypersonic-license.txt` - License agreement to use Hypersonic database.
3. `lib/jakarta-commons-license.txt` - License agreement to use Jakarta Commons libraries.
4. `lib/serp-license.txt` - License agreement to use Serp libraries.
5. `lib/sun-treetable-license.txt` - License agreement to use Sun's treetable.
6. `lib/xanlan-license.txt` - License agreement to use Apache Xanlan Project software.
7. `lib/xerces-license.txt` - License agreement to use Apache Xerces Project software.

2.2.3. Quick Start

Follow the instructions in the appropriate installation section below. Note that if you are reading this in the `README.txt` file, the online documentation can be found at: <http://e-docs.bea.com/kodo/docs40/index.html>

If you received this package on a CD, you will need to download a valid license key from: <http://ftpmain.bea.com/download/pub/license/kodo/kodo40/license.bea> While obtaining your license key, we highly recommend that you download the latest version of Kodo.

2.2.4. Upgrading from Previous Kodo Versions

There may be significant differences to properties, mappings, and internal APIs between major releases of Kodo. For information about how to upgrade from a previous Kodo versions, see Appendix D of the Kodo Reference Guide.

2.2.5. Requirements

- A valid Kodo license key. Evaluation keys are available at: <http://ftpmain.bea.com/download/pub/license/kodo/kodo40/license.bea>
- JDK 1.5 or higher for JPA; JDK 1.2 or greater for JDO.
- A relational database with JDBC driver, such as Oracle, IBM DB2, Microsoft SQLServer, Sybase, Pointbase, MySQL, PostgreSQL, or Hypersonic SQL. This installation is bundled with Hypersonic, which requires no installation and minimal configuration.

2.2.6. Terminology

KODOHOME

This is the directory where you installed Kodo.

JDKHOME

This is the directory where you installed the JDK.

2.2.7. Windows Installation

1. Read and agree to any third party license agreements.
2. Edit `KODOHOME/bin/kodocmd.bat` and `KODOHOME/bin/kodocommand.bat` so that `KODODIR` and `JDKHOME` are set correctly.
3. Run `KODOHOME/bin/kodocmd.bat` or `KODOHOME/bin/kodocommand.bat`. (The former relies on 'cmd', the command shell for Windows XP, 2000, and NT; the latter uses 'command', the command shell for Windows ME, 98, and 95.) All tutorial commands should be executed from this shell.
4. Open <http://e-docs.bea.com/kodo/docs40/index.html> For JPA, navigate to the **Kodo JPA Tutorials**. For JDO, navigate to the **Kodo JDO Tutorials**. These tutorials are in Part IV of the Kodo documentation.
5. Change to the `KODOHOME/tutorial/persistence` directory or the `KODOHOME/tutorial/jdo` directory.
6. Start the tutorial.

2.2.8. POSIX (Linux, Solaris, Mac OS X, Windows with cygwin, etc.) Installation

1. Open a shell.
2. Ensure that your `CLASSPATH` environment variable contains the base Java runtime package (`$JDKHOME/jre/lib/rt.jar` for JDK 1.2 or higher).
3. Change to the `KODOHOME` directory.
4. Type `'chmod a+x bin/*'`. This will give you execute permissions on all packaged shell scripts.
5. Type `'source bin/envsetup'` (On Windows with Cygwin, type `'source bin/cygsetup'`). This will modify your

CLASSPATH and PATH environment variables to add in the libraries in KODOHOME/lib and the executables in KODOHOME/bin.

6. Open <http://e-docs.bea.com/kodo/docs40/index.html> For JPA, navigate to the **Kodo JPA Tutorials**. For JDO, navigate to the **Kodo JDO Tutorials**. These tutorials are in Part IV of the Kodo documentation.
7. Change to the KODOHOME/tutorial/persistence directory or the KODOHOME/tutorial/jdo directory.
8. Start the tutorial.

2.2.9. Common Installation Problems

Problem: Shell error when running '`javac *.java`' or when using '`java`'

Solution: Ensure that `java` and `javac` are installed and in your path. To verify that they are installed, type '`java`' on a line by itself. If this returns a usage statement, then `java` is installed and in your path. Repeat with '`javac`' instead of '`java`' to see if `javac` is installed correctly. If these tests fail, install JDK 1.2 or greater. Also, please make sure that `kodocmd.bat` sets up your path properly.

Problem: When running '`kodoc`', you get a `NoClassDefFoundError` with a message like the following:
`java.lang.NoClassDefFoundError: Animal (wrong name: tutorial/persistence/Animal)`

Solution: This often means that you are invoking `kodoc` from the directory that contains the class `Animal`, and '.' is in your classpath. Fixing your classpath or running `kodoc` from a different directory should solve this problem.

Problem: A 'sealing violation' occurs when running the enhancer.

Solution: This indicates that some other library in your CLASSPATH is conflicting with the jars packaged with this distribution.

2.2.10. Resources

If you have any technical questions while evaluating or installing Kodo, please send an email to support@solarmetric.com.

The web site (<http://www.solarmetric.com>) has Kodo development resources, including frequently asked questions, our bug tracking system, Kodo newsgroups, access to technical support, and links to a variety of technical articles and third party tutorials.

Part 2. Java Persistence API

Table of Contents

1. Introduction	12
1.1. Intended Audience	12
1.2. Lightweight Persistence	12
2. Why JPA?	13
3. EJB Persistence Architecture	15
3.1. EJB Exceptions	16
4. Entity	18
4.1. Restrictions on Persistent Classes	19
4.1.1. Default or No-Arg Constructor	19
4.1.2. Final	19
4.1.3. Identity Fields	19
4.1.4. Version Field	19
4.1.5. Inheritance	20
4.1.6. Persistent Fields	20
4.1.7. Conclusions	21
4.2. Entity Identity	21
4.2.1. Identity Class	22
4.2.1.1. Identity Hierarchies	23
4.3. Lifecycle Callbacks	24
4.3.1. Callback Methods	24
4.3.2. Using Callback Methods	25
4.3.3. Using Entity Listeners	26
4.3.4. Entity Listeners Hierarchy	26
4.4. Conclusions	27
5. Metadata	28
5.1. Class Metadata	29
5.1.1. Entity	29
5.1.2. Id Class	30
5.1.3. Mapped Superclass	30
5.1.4. Embeddable	30
5.1.5. EntityListeners	31
5.1.6. Example	31
5.2. Field and Property Metadata	33
5.2.1. Transient	34
5.2.2. Id	34
5.2.3. Generated Value	34
5.2.4. Embedded Id	35
5.2.5. Version	35
5.2.6. Basic	35
5.2.6.1. Fetch Type	36
5.2.7. Embedded	36
5.2.8. Many-To-One	36
5.2.8.1. Cascade Type	37
5.2.9. One-To-Many	38
5.2.9.1. Bidirectional Relations	39
5.2.10. One-To-One	39
5.2.11. Many-To-Many	40
5.2.12. Order By	41
5.2.13. Map Key	41
5.2.14. Persistent Field Defaults	41
5.3. XML Schema	42
5.4. Conclusion	58
6. Persistence	63

6.1. persistence.xml	63
6.2. Non-EE Use	65
7. EntityManagerFactory	66
7.1. Obtaining an EntityManagerFactory	66
7.2. Obtaining EntityManagers	66
7.3. Persistence Context	67
7.3.1. Transaction Persistence Context	67
7.3.2. Extended Persistence Context	68
7.4. Closing the EntityManagerFactory	69
8. EntityManager	70
8.1. Transaction Association	70
8.2. Entity Lifecycle Management	71
8.3. Lifecycle Examples	74
8.4. Entity Identity Management	76
8.5. Cache Management	76
8.6. Query Factory	77
8.7. Closing	78
9. Transaction	79
9.1. Transaction Types	79
9.2. The EntityManagerFactory Interface	80
10. JPA Query	82
10.1. JPQL API	82
10.1.1. Query Basics	82
10.1.2. Relation Traversal	85
10.1.3. Fetch Joins	86
10.1.4. JPQL Functions	86
10.1.5. Polymorphic Queries	88
10.1.6. Query Parameters	88
10.1.7. Ordering	89
10.1.8. Aggregates	89
10.1.9. Named Queries	90
10.1.10. Delete By Query	90
10.1.11. Update By Query	91
10.2. JPQL Language Reference	91
10.2.1. JPQL Statement Types	92
10.2.1.1. JPQL Select Statement	92
10.2.1.2. JPQL Update and Delete Statements	92
10.2.2. JPQL Abstract Schema Types and Query Domains	92
10.2.2.1. JPQL Entity Naming	93
10.2.2.2. JPQL Schema Example	93
10.2.3. JPQL FROM Clause and Navigational Declarations	94
10.2.3.1. JPQL FROM Identifiers	94
10.2.3.2. JPQL Identification Variables	96
10.2.3.3. JPQL Range Declarations	96
10.2.3.4. JPQL Path Expressions	97
10.2.3.5. JPQL Joins	98
10.2.3.5.1. JPQL Inner Joins (Relationship Joins)	98
10.2.3.5.2. JPQL Outer Joins	98
10.2.3.5.3. JPQL Fetch Joins	99
10.2.3.6. JPQL Collection Member Declarations	99
10.2.3.7. JPQL Polymorphism	100
10.2.4. JPQL WHERE Clause	100
10.2.5. JPQL Conditional Expressions	100
10.2.5.1. JPQL Literals	100
10.2.5.2. JPQL Identification Variables	101
10.2.5.3. JPQL Path Expressions	101
10.2.5.4. JPQL Input Parameters	101
10.2.5.4.1. JPQL Positional Parameters	101
10.2.5.4.2. JPQL Named Parameters	101

10.2.5.5. JPQL Conditional Expression Composition	101
10.2.5.6. JPQL Operators and Operator Precedence	102
10.2.5.7. JPQL Between Expressions	102
10.2.5.8. JPQL In Expressions	103
10.2.5.9. JPQL Like Expressions	104
10.2.5.10. JPQL Null Comparison Expressions	104
10.2.5.11. JPQL Empty Collection Comparison Expressions	104
10.2.5.12. JPQL Collection Member Expressions	105
10.2.5.13. JPQL Exists Expressions	105
10.2.5.14. JPQL All or Any Expressions	105
10.2.5.15. JPQL Subqueries	106
10.2.5.16. JPQL Functional Expressions	106
10.2.5.16.1. JPQL String Functions	107
10.2.5.16.2. JPQL Arithmetic Functions	107
10.2.5.16.3. JPQL Datetime Functions	107
10.2.6. JPQL GROUP BY, HAVING	107
10.2.7. JPQL SELECT Clause	108
10.2.7.1. JPQL Result Type of the SELECT Clause	109
10.2.7.2. JPQL Constructor Expressions	109
10.2.7.3. JPQL Null Values in the Query Result	109
10.2.7.4. JPQL Aggregate Functions	109
10.2.7.4.1. JPQL Aggregate Examples	110
10.2.8. JPQL ORDER BY Clause	110
10.2.9. JPQL Bulk Update and Delete	111
10.2.10. JPQL Null Values	111
10.2.11. JPQL Equality and Comparison Semantics	112
10.2.12. JPQL BNF	112
11. SQL Queries	116
11.1. Creating SQL Queries	116
11.2. Retrieving Persistent Objects with SQL	116
12. Mapping Metadata	118
12.1. Table	119
12.2. Unique Constraints	122
12.3. Column	123
12.4. Identity Mapping	123
12.5. Generators	126
12.5.1. Sequence Generator	126
12.5.2. TableGenerator	127
12.5.3. Example	128
12.6. Inheritance	129
12.6.1. Single Table	130
12.6.1.1. Advantages	131
12.6.1.2. Disadvantages	131
12.6.2. Joined	131
12.6.2.1. Advantages	133
12.6.2.2. Disadvantages	133
12.6.3. Table Per Class	134
12.6.3.1. Advantages	135
12.6.3.2. Disadvantages	135
12.6.4. Putting it All Together	135
12.7. Discriminator	138
12.8. Field Mapping	143
12.8.1. Basic Mapping	143
12.8.1.1. LOBs	143
12.8.1.2. Enumerated	143
12.8.1.3. Temporal Types	144
12.8.1.4. The Updated Mappings	144
12.8.2. Secondary Tables	149
12.8.3. Embedded Mapping	150

12.8.4. Direct Relations	153
12.8.5. Join Table	156
12.8.6. Bidirectional Mapping	158
12.8.7. Map Mapping	158
12.9. The Complete Mappings	160
13. Conclusion	168

Chapter 1. Introduction

Enterprise Java Beans 3.0 Persistence (EJB persistence) is a specification from Sun Microsystems for the persistence of Java objects to any relational datastore. EJB persistence requires J2SE 1.5 (also referred to as "Java 5") or higher, as it makes heavy use of new Java language features such as annotations and generics. This document provides an overview of EJB persistence. Unless otherwise noted, the information presented applies to all EJB persistence implementations.

Note

This document describes the Public Draft of the EJB 3.0 persistence specification.

For coverage of Kodo's many extensions to the EJB persistence specification, see the [Reference Guide](#).

1.1. Intended Audience

This document is intended for developers who want to learn about EJB persistence in order to use it in their applications. It assumes that you have a strong knowledge of object-oriented concepts and Java, including Java 5 annotations and generics. It also assumes some experience with relational databases and the Structured Query Language (SQL).

1.2. Lightweight Persistence

Persistent data is information that can outlive the program that creates it. The majority of complex programs use persistent data: GUI applications need to store user preferences across program invocations, web applications track user movements and orders over long periods of time, etc.

Lightweight persistence is the storage and retrieval of persistent data with little or no work from you, the developer. For example, Java serialization is a form of lightweight persistence because it can be used to persist Java objects directly to a file with very little effort. The capabilities of serialization as a lightweight persistence mechanism pale, however, in comparison to those provided by EJB. The next chapter compares EJB to serialization and other available persistence mechanisms.

Chapter 2. Why JPA?

Java developers who need to store and retrieve persistent data already have several options available to them: serialization, JDBC, JDO, proprietary object-relational mapping tools, object databases, and EJB 2 entity beans. Why introduce yet another persistence framework? The answer to this question is that with the exception of JDO, each of the aforementioned persistence solutions has severe limitations. JPA attempts to overcome these limitations, as illustrated by the table below.

Table 2.1. Persistence Mechanisms

Supports:	Serialization	JDBC	ORM	ODB	EJB 2	JDO	JPA
Java Objects	Yes	No	Yes	Yes	Yes	Yes	Yes
Advanced OO Concepts	Yes	No	Yes	Yes	No	Yes	Yes
Transactional Integrity	No	Yes	Yes	Yes	Yes	Yes	Yes
Concurrency	No	Yes	Yes	Yes	Yes	Yes	Yes
Large Data Sets	No	Yes	Yes	Yes	Yes	Yes	Yes
Existing Schema	No	Yes	Yes	No	Yes	Yes	Yes
Relational and Non-Relational Stores	No	No	No	No	Yes	Yes	No
Queries	No	Yes	Yes	Yes	Yes	Yes	Yes
Strict Standards / Portability	Yes	No	No	No	Yes	Yes	Yes
Simplicity	Yes	Yes	Yes	Yes	No	Yes	Yes

- *Serialization* is Java's built-in mechanism for transforming an object graph into a series of bytes, which can then be sent over the network or stored in a file. Serialization is very easy to use, but it is also very limited. It must store and retrieve the entire object graph at once, making it unsuitable for dealing with large amounts of data. It cannot undo changes that are made to objects if an error occurs while updating information, making it unsuitable for applications that require strict data integrity. Multiple threads or programs cannot read and write the same serialized data concurrently without conflicting with each other. It provides no query capabilities. All these factors make serialization useless for all but the most trivial persistence needs.
- Many developers use the *Java Database Connectivity* (JDBC) APIs to manipulate persistent data in relational databases. JDBC overcomes most of the shortcomings of serialization: it can handle large amounts of data, has mechanisms to ensure data integrity, supports concurrent access to information, and has a sophisticated query language in SQL. Unfortunately, JDBC does not duplicate serialization's ease of use. The relational paradigm used by JDBC was not designed for storing objects, and therefore forces you to either abandon object-oriented programming for the portions of your code that deal with persistent data, or to find a way of mapping object-oriented concepts like inheritance to relational databases yourself.
- There are many proprietary software products that can perform the mapping between objects and relational database tables for you. These *object-relational mapping* (ORM) frameworks allow you to focus on the object model and not concern yourself with the mismatch between the object-oriented and relational paradigms. Unfortunately, each of these product has its own set of APIs. Your code becomes tied to the proprietary interfaces of a single vendor. If the vendor raises prices, fails to fix show-stopping bugs, or falls behind in features, you cannot switch to another product without rewriting all of your persistence code. This is referred to as vendor lock-in.

- Rather than map objects to relational databases, some software companies have developed a form of database designed specifically to store objects. These *object databases* (ODBs) are often much easier to use than object-relational mapping software. The Object Database Management Group (ODMG) was formed to create a standard API for accessing object databases; few object database vendors, however, comply with the ODMG's recommendations. Thus, vendor lock-in plagues object databases as well. Many companies are also hesitant to switch from tried-and-true relational systems to the relatively unknown object database technology. Fewer data-analysis tools are available for object database systems, and there are vast quantities of data already stored in older relational databases. For all of these reasons and more, object databases have not caught on as well as their creators hoped.
- The Enterprise Edition of the Java platform introduced entity Enterprise Java Beans (EJBs). EJB 2.x entities are components that represent persistent information in a datastore. Like object-relational mapping solutions, EJB 2.x entities provide an object-oriented view of persistent data. Unlike object-relational software, however, EJB 2.x entities are not limited to relational databases; the persistent information they represent may come from an Enterprise Information System (EIS) or other storage device. Also, EJB 2.x entities use a strict standard, making them portable across vendors. Unfortunately, the EJB 2.x standard is somewhat limited in the object-oriented concepts it can represent. Advanced features like inheritance, polymorphism, and complex relations are absent. Additionally, EJB 2.x entities are difficult to code, and they require heavyweight and often expensive application servers to run.
- The JDO specification uses an API that is strikingly similar to JPA. JDO, however, supports non-relational databases, a feature that some argue dilutes the specification.

JPA combines the best features from each of the persistence mechanisms listed above. Creating entities under JPA is as simple as creating serializable classes. JPA supports the large data sets, data consistency, concurrent use, and query capabilities of JDBC. Like object-relational software and object databases, JPA allows the use of advanced object-oriented concepts such as inheritance. JPA avoids vendor lock-in by relying on a strict specification like JDO and EJB 2.x entities. JPA focuses on relational databases. And like JDO, JPA is extremely easy to use.

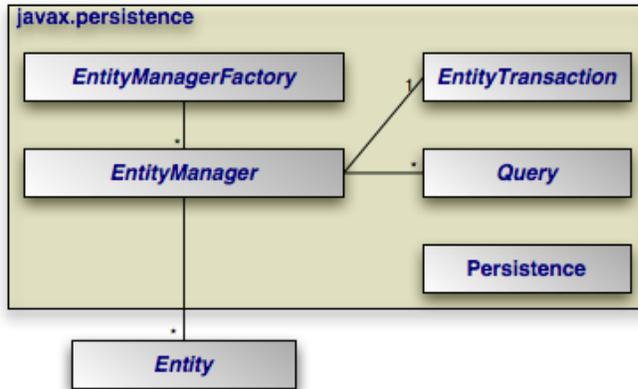
Note

Kodo typically stores data in relational databases, but can be customized for use with non-relational datastores as well.

JPA is not ideal for every application. For many applications, though, it provides an exciting alternative to other persistence mechanisms.

Chapter 3. EJB Persistence Architecture

The diagram below illustrates the relationships between the primary components of the EJB architecture.



Note

A number of the depicted interfaces are only required outside of an EJB3-compliant application server. In an application server, `EntityManager` instances are typically injected, rendering the `EntityManagerFactory` unnecessary. Also, transactions within an application server are handled using standard application server transaction controls. Thus, the `EntityTransaction` also goes unused.

- **Persistence:** The `javax.persistence.Persistence` class contains static helper methods to obtain `EntityManagerFactory` instances in a vendor-neutral fashion.
- **EntityManagerFactory:** The `javax.persistence.EntityManagerFactory` class is a factory for `EntityManager`s.
- **EntityManager:** The `javax.persistence.EntityManager` is the primary EJB persistence interface used by applications. Each `EntityManager` manages a set of persistent objects, and has APIs to insert new objects and delete existing ones. When used outside the container, there is a one-to-one relationship between an `EntityManager` and an `EntityTransaction`. `EntityManager`s also act as factories for `Query` instances.
- **Entity:** Entities are persistent objects that represent datastore records.
- **EntityTransaction:** Each `EntityManager` has a one-to-one relation with a single `javax.persistence.EntityTransaction`. `EntityTransactions` allow operations on persistent data to be grouped into units of work that either completely succeed or completely fail, leaving the datastore in its original state. These all-or-nothing operations are important for maintaining data integrity.
- **Query:** The `javax.persistence.Query` interface is implemented by each EJB vendor to find persistent objects that meet certain criteria. EJB standardizes support for queries using both the EJB Query Language (JPQL) and the Structured Query Language (SQL). You obtain `Query` instances from an `EntityManager`.

The example below illustrates how the EJB interfaces interact to execute a JPQL query and update persistent objects. The example assumes execution outside a container.

Example 3.1. Interaction of Interfaces Outside Container

```
// get an EntityManagerFactory using the Persistence class; typically
// the factory is cached for easy repeated use
EntityManagerFactory factory = Persistence.createEntityManagerFactory (null);

// get an EntityManager from the factory
EntityManager em = factory.createEntityManager (PersistenceContextType.EXTENDED);

// updates take place within transactions
EntityTransaction tx = em.getTransaction ();
tx.begin ();

// query for all employees who work in our research division
// and put in over 40 hours a week average
Query query = em.createQuery ("select e from Employee e where "
    + "e.division.name = 'Research' AND e.avgHours > 40");
List results = query.getResultList ();

// give all those hard-working employees a raise
for (Object res : results)
{
    Employee emp = (Employee) res;
    emp.setSalary (emp.getSalary () * 1.1);
}

// commit the updates and free resources
tx.commit ();
em.close ();
factory.close ();
```

Within a container, the `EntityManager` will be injected and transactional handled declaratively. Thus, the in-container version of the example consists entirely of business logic:

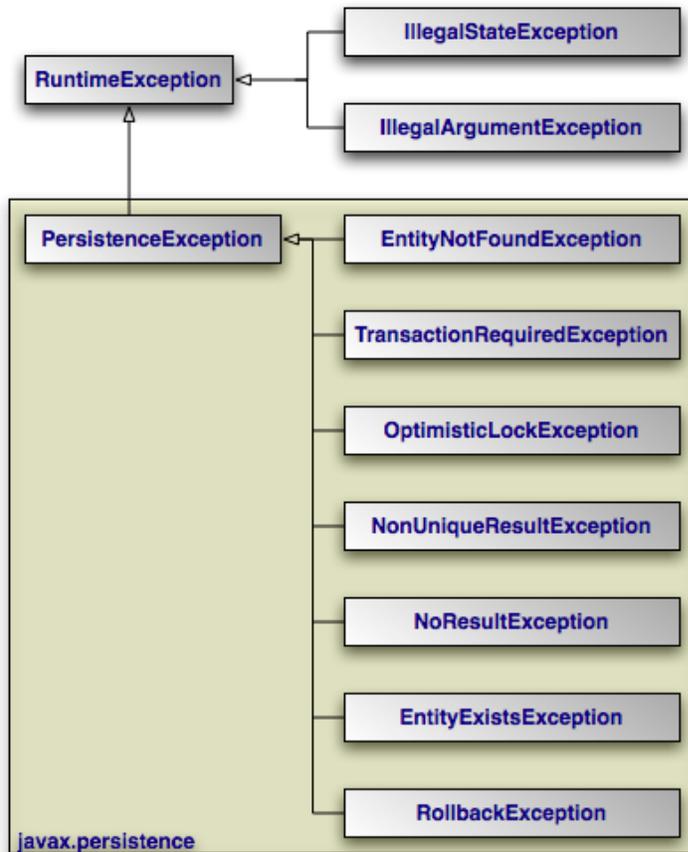
Example 3.2. Interaction of Interfaces Inside Container

```
// query for all employees who work in our research division
// and put in over 40 hours a week average - note that the EntityManager em
// is injected using a @Resource annotation
Query query = em.createQuery ("select e from Employee e where "
    + "e.division.name = 'Research' and e.avgHours > 40");
List results = query.getResultList ();

// give all those hard-working employees a raise
for (Object res : results)
{
    emp = (Employee) res;
    emp.setSalary (emp.getSalary () * 1.1);
}
```

The remainder of this document explores the EJB interfaces in detail. We present them in roughly the order that you will use them as you develop your application.

3.1. EJB Exceptions



The diagram above depicts the EJB persistence exception architecture. All exceptions are unchecked. EJB persistence uses standard exceptions where appropriate, most notably `IllegalArgumentException`s and `IllegalStateException`s. The specification also provides a few EJB-specific exceptions in the `javax.persistence` package. These exceptions should be self-explanatory. See the **Javadoc** for additional details on EJB exceptions.

Note

All exceptions thrown by Kodo implement `kodo.util.ExceptionInfo` to provide you with additional error information.

Chapter 4. Entity

JPA recognizes two types of persistent classes: *entity* classes and *embeddable* classes. Each persistent instance of an entity class - each *entity* - represents a unique datastore record. You can use the `EntityManager` to find an entity by its persistent identity (covered later in this chapter), or use a `Query` to find entities matching certain criteria.

An instance of an embeddable class, on the other hand, is only stored as part of a separate entity. Embeddable instances have no persistent identity, and are never returned directly from the `EntityManager` or from a `Query`.

Despite these differences, there are few differences between entity classes and embeddable classes. In fact, writing either type of persistent class is little different than writing any other class. There are no special parent classes to extend from, field types to use, or methods to write. This is one important way in which JPA makes persistence transparent to you, the developer.

Note

JPA supports both fields and JavaBean properties as persistent state. For simplicity, however, we will refer to all persistent state as persistent fields, unless we want to note a unique aspect of persistent properties.

Example 4.1. Persistent Class

```
package org.mag;

/**
 * Example persistent class. Notice that it looks exactly like any other
 * class. JPA makes writing persistent classes completely transparent.
 */
public class Magazine
{
    private String    isbn;
    private String    title;
    private Set       articles = new HashSet ();
    private Article   coverArticle;
    private int       copiesSold;
    private double    price;
    private Company   publisher;
    private int       version;

    protected Magazine ()
    {
    }

    public Magazine (String title, String isbn)
    {
        this.title = title;
        this.isbn = isbn;
    }

    public void publish (Company publisher, double price)
    {
        this.publisher = publisher;
        publisher.addMagazine (this);
        this.price = price;
    }

    public void sell ()
    {
        copiesSold++;
        publisher.addRevenue (price);
    }

    public void addArticle (Article article)
    {
        articles.add (article);
    }

    // rest of methods omitted
}
```

4.1. Restrictions on Persistent Classes

There are very few restrictions placed on persistent classes. Still, it never hurts to familiarize yourself with exactly what JPA does and does not support.

4.1.1. Default or No-Arg Constructor

The JPA specification requires that all persistent classes have a no-arg constructor. This constructor may be public or protected. Because the compiler automatically creates a default no-arg constructor when no other constructor is defined, only classes that define constructors must also include a no-arg constructor.

Note

Kodo's *enhancer* will automatically add a protected no-arg constructor to your class when required. Therefore, this restriction does not apply when using Kodo. See [Section 5.2, “Enhancement” \[452\]](#) of the Reference Guide for details.

4.1.2. Final

Entity classes may not be final. No method of an entity class can be final.

Note

Kodo supports final classes and final methods.

4.1.3. Identity Fields

All entity classes must declare one or more fields which together form the persistent identity of an instance. These are called *identity* or *primary key* fields. In our `Magazine` class, `isbn` and `title` are identity fields, because no two magazine records in the datastore can have the same `isbn` and `title` values. [Section 5.2.2, “Id” \[34\]](#) will show you how to denote your identity fields in JPA metadata. [Section 4.2, “Entity Identity” \[21\]](#) below examines persistent identity.

Note

Kodo fully supports identity fields, but does not require them. See [Section 5.3, “Object Identity” \[455\]](#) of the Reference Guide for details.

4.1.4. Version Field

The `version` field in our `Magazine` class may seem out of place. JPA uses a version field in your entity to detect concurrent modifications to the same datastore record. When the JPA runtime detects an attempt to concurrently modify the same record, it throws an exception to the transaction attempting to commit last. This prevents overwriting the previous commit with stale data.

The version field is not required, but without one concurrent threads or processes might succeed in making conflicting changes to the same record at the same time. This is unacceptable to most applications. [Section 5.2.5, “Version” \[35\]](#) shows you how to designate a version field in JPA metadata.

The version field must be an integral type (`int`, `Long`, etc) or a `java.sql.Timestamp`. You should consider version fields immutable. Changing the field value has undefined results.

Note

Kodo fully supports version fields, but does not require them for concurrency detection. Kodo can maintain surrogate version values or use state comparisons to detect concurrent modifications. See [Section 7.7, “Additional JPA Mappings” \[508\]](#) in the Reference Guide.

4.1.5. Inheritance

JPA fully supports inheritance in persistent classes. It allows persistent classes to inherit from non-persistent classes, persistent classes to inherit from other persistent classes, and non-persistent classes to inherit from persistent classes. It is even possible to form inheritance hierarchies in which persistence skips generations. There are, however, a few important limitations:

- Persistent classes cannot inherit from certain natively-implemented system classes such as `java.net.Socket` and `java.lang.Thread`.
- If a persistent class inherits from a non-persistent class, the fields of the non-persistent superclass cannot be persisted.
- All classes in an inheritance tree must use the same identity type. We cover entity identity in [Section 4.2, “Entity Identity” \[?\]](#).

4.1.6. Persistent Fields

JPA manages the state of all persistent fields. Before you access persistent state, the JPA runtime makes sure that it has been loaded from the datastore. When you set a field, the runtime records that it has changed so that the new value will be persisted. This allows you to treat the field in exactly the same way you treat any other field - another aspect of JPA's transparency.

JPA does not support static or final fields. It does, however, include built-in support for most common field types. These types can be roughly divided into three categories: immutable types, mutable types, and relations.

Immutable types, once created, cannot be changed. The only way to alter a persistent field of an immutable type is to assign a new value to the field. JPA supports the following immutable types:

- All primitives (`int`, `float`, `byte`, etc)
- All primitive wrappers (`java.lang.Integer`, `java.lang.Float`, `java.lang.Byte`, etc)
- `java.lang.String`
- `java.math.BigInteger`
- `java.math.BigDecimal`

JPA also supports `byte[]` and `char[]` as immutable types. That is, you can persist fields of these types, but you should not manipulate individual array indexes without resetting the array into the persistent field.

Persistent fields of *mutable* types can be altered without assigning the field a new value. Mutable types can be modified directly through their own methods. The JPA specification requires that implementations support the following mutable field types:

- `java.util.Date`
- `java.util.Calendar`
- `java.sql.Date`
- `java.sql.Timestamp`

- Enums
- Entity types (relations between entities)
- Embeddable types
- `java.util.Collections` of entities
- `java.util.Sets` of entities
- `java.util.Lists` of entities
- `java.util.Maps` in which each entry maps the value of one of an entity's fields to that entity.

Collection and map types may be parameterized.

Most JPA implementations also have support for persisting serializable values as binary data in the datastore. **Chapter 5, Metadata [28]** has more information on persisting serializable types.

Note

Kodo also supports arrays, `java.lang.Number`, `java.util.Locale`, all JDK 1.2 `Set`, `List`, and `Map` types, collections and maps of immutable and embedded as well as entity types, and many other mutable and immutable field types. Kodo also allows you to plug in support for custom types.

4.1.7. Conclusions

This section detailed all of the restrictions JPA places on persistent classes. While it may seem like we presented a lot of information, you will seldom find yourself hindered by these restrictions in practice. Additionally, there are often ways of using JPA's other features to circumvent any limitations you run into.

4.2. Entity Identity

Java recognizes two forms of object identity: numeric identity and qualitative identity. If two references are *numerically* identical, then they refer to the same JVM instance in memory. You can test for this using the `==` operator. *Qualitative* identity, on the other hand, relies on some user-defined criteria to determine whether two objects are "equal". You test for qualitative identity using the `equals` method. By default, this method simply relies on numeric identity.

JPA introduces another form of object identity called *entity identity* or *persistent identity*. Entity identity tests whether two persistent objects represent the same state in the datastore.

The entity identity of each persistent instance is encapsulated in its *identity field(s)*. If two entities of the same type have the same identity field values, then the two entities represent the same state in the datastore. Each entity's identity field values must be unique among all other entities of the same type.

Identity fields must be primitives, primitive wrappers, `Strings`, `Dates`, `Timestamps`, or embeddable types. Notably, other entities instances can *not* be used as identity fields.

Note

For legacy schemas with binary primary key columns, Kodo also supports using identity fields of type `byte[]`. When you use a `byte[]` identity field, you must create an identity class. Identity classes are covered below.

Warning

Changing the fields of an embeddable instance while it is assigned to an identity field has undefined results. Always treat embeddable identity instances as immutable objects in your applications.

If you are dealing with a single persistence context (see [Section 7.3, “Persistence Context” \[67\]](#)), then you do not have to compare identity fields to test whether two entity references represent the same state in the datastore. There is a much easier way: the `==` operator. JPA requires that each persistence context maintain only one JVM object to represent each unique datastore record. Thus, entity identity is equivalent to numeric identity within a persistence context. This is referred to as the *uniqueness requirement*.

The uniqueness requirement is extremely important - without it, it would be impossible to maintain data integrity. Think of what could happen if two different objects in the same transaction were allowed to represent the same persistent data. If you made different modifications to each of these objects, which set of changes should be written to the datastore? How would your application logic handle seeing two different "versions" of the same data? Thanks to the uniqueness requirement, these questions do not have to be answered.

4.2.1. Identity Class

If your entity has only one identity field, you can use the value of that field as the entity's identity object in all `EntityManager` APIs. Otherwise, you must supply an identity class to use for identity objects. Your identity class must meet the following criteria:

- The class must be public.
- The class must be serializable.
- The class must have a public no-args constructor.
- The names of the non-static fields or properties of the class must be the same as the names of the identity fields or properties of the corresponding entity class, and the types must be identical.
- The `equals` and `hashCode` methods of the class must use the values of all fields or properties corresponding to identity fields or properties in the entity class.
- If the class is an inner class, it must be `static`.
- All entity classes related by inheritance must use the same identity class, or else each entity class must have its own identity class whose inheritance hierarchy mirrors the inheritance hierarchy of the owning entity classes (see [Section 4.2.1.1, “Identity Hierarchies” \[23\]](#)).

Note

Though you may still create identity classes by hand, Kodo provides the `appidtool` to automatically generate proper identity classes based on your identity fields. See [Section 5.3.2, “Application Identity Tool” \[456\]](#) of the Reference Guide.

Example 4.2. Identity Class

This example illustrates a proper identity class for an entity with multiple identity fields.

```
/**
```

```

* Persistent class using application identity.
*/
public class Magazine
{
    private String isbn;    // identity field
    private String title;  // identity field

    // rest of fields and methods omitted

    /**
     * Application identity class for Magazine.
     */
    public static class MagazineId
    {
        // each identity field in the Magazine class must have a
        // corresponding field in the identity class
        public String isbn;
        public String title;

        /**
         * Equality must be implemented in terms of identity field
         * equality, and must use instanceof rather than comparing
         * classes directly (some JPA implementations may subclass the
         * identity class).
         */
        public boolean equals (Object other)
        {
            if (other == this)
                return true;
            if (!(other instanceof MagazineId))
                return false;

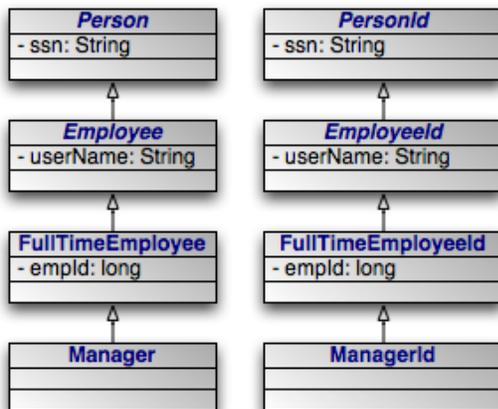
            MagazineId mi = (MagazineId) other;
            return (isbn == mi.isbn
                || (isbn != null && isbn.equals (mi.isbn)))
                && (title == mi.title
                || (title != null && title.equals (mi.title)));
        }

        /**
         * Hashcode must also depend on identity values.
         */
        public int hashCode ()
        {
            return ((isbn == null) ? 0 : isbn.hashCode ())
                ^ ((title == null) ? 0 : title.hashCode ());
        }

        public String toString ()
        {
            return isbn + ":" + title;
        }
    }
}

```

4.2.1.1. Identity Hierarchies



An alternative to having a single identity class for an entire inheritance hierarchy is to have one identity class per level in the inheritance hierarchy. The requirements for using a hierarchy of identity classes are as follows:

- The inheritance hierarchy of identity classes must exactly mirror the hierarchy of the persistent classes that they identify. In the example pictured above, abstract class `Person` is extended by abstract class `Employee`, which is extended by non-abstract class `FullTimeEmployee`, which is extended by non-abstract class `Manager`. The corresponding identity classes, then, are an abstract `PersonId` class, extended by an abstract `EmployeeId` class, extended by a non-abstract `FullTimeEmployeeId` class, extended by a non-abstract `ManagerId` class.
- Subclasses in the identity hierarchy may define additional identity fields until the hierarchy becomes non-abstract. In the aforementioned example, `Person` defines an identity field `ssn`, `Employee` defines additional identity field `userName`, and `FullTimeEmployee` adds a final identity field, `empId`. However, `Manager` may not define any additional identity fields, since it is a subclass of a non-abstract class. The hierarchy of identity classes, of course, must match the identity field definitions of the persistent class hierarchy.
- It is not necessary for each abstract class to declare identity fields. In the previous example, the abstract `Person` and `Employee` classes could declare no identity fields, while the first concrete subclass, `FullTimeEmployee`, could define one or more identity fields.
- All subclasses of a concrete identity class must be `equals` and `hashCode`-compatible with the concrete superclass. This means that in our example, a `ManagerId` instance and a `FullTimeEmployeeId` instance with the same identity field values should have the same hash code, and should compare equal to each other using the `equals` method of either one. In practice, this requirement reduces to the following coding practices:
 1. Use `instanceof` instead of comparing `Class` objects in the `equals` methods of your identity classes.
 2. An identity class that extends another non-abstract identity class should not override `equals` or `hashCode`.

4.3. Lifecycle Callbacks

It is often necessary to perform various actions at different stages of a persistent object's lifecycle. JPA includes a variety of callback methods for monitoring changes in the lifecycle of your persistent objects. These callbacks can be defined on the persistent classes themselves and on non-persistent listener classes.

4.3.1. Callback Methods

Every persistence event has a corresponding callback method marker. These markers are shared between persistent classes and their listeners. You can use these markers to designate a method for callback either by annotating that method or by listing the method in the XML mapping file for a given class. The lifecycle events and their corresponding method markers are:

- **PrePersist:** Methods marked with this annotation will be invoked before an object is persisted. This could be used for assigning primary key values to persistent objects. This is equivalent to the XML element tag `pre-persist`.
- **PostPersist:** Methods marked with this annotation will be invoked after an object has transitioned to the persistent state. You might want to use such methods to update a screen after a new row is added. This is equivalent to the XML element tag `post-persist`.
- **PostLoad:** Methods marked with this annotation will be invoked after all eagerly fetched fields of your class have been loaded from the datastore. No other persistent fields can be accessed in this method. This is equivalent to the XML element tag `post-load`.

`PostLoad` is often used to initialize non-persistent fields whose values depend on the values of persistent fields, such as a complex datastructure.

- **PreUpdate:** Methods marked with this annotation will be invoked just before the persistent values in your objects are

flushed to the datastore. This is equivalent to the XML element tag `pre-update`.

`PreUpdate` is the complement to `PostLoad`. While methods marked with `PostLoad` are most often used to initialize non-persistent values from persistent data, methods annotated with `PreUpdate` is normally used to set persistent fields with information cached in non-persistent data.

- **PostUpdate:** Methods marked with this annotation will be invoked after changes to a given instance have been stored in the datastore. This is useful for clearing stale data cached at the application layer. This is equivalent to the XML element tag `post-update`.
- **PreRemove:** Methods marked with this annotation will be invoked before an object transactions to the deleted state. Access to persistent fields is valid within this method. You might use this method to cascade the deletion to related objects based on complex criteria, or to perform other cleanup. This is equivalent to the XML element tag `pre-remove`.
- **PostRemove:** Methods marked with this annotation will be invoked after an object has been marked for deletion. This is equivalent to the XML element tag `post-remove`.

4.3.2. Using Callback Methods

When declaring callback methods on a persistent class, any method may be used which takes no arguments and is not shared with any property access fields. Multiple events can be assigned to a single method as well.

Below is an example of how to declare callback methods on persistent classes:

```
/**
 * Example persistent class declaring our entity listener.
 */
@Entity
public class Magazine
{
    @Transient
    private byte[][] data;

    @ManyToMany
    private List<Photo> photos;

    @PostLoad
    public void convertPhotos ()
    {
        data = new byte[photos.size ()][];
        for (int i = 0; i < photos.size (); i++)
            data[i] = photos.get (i).toByteArray ();
    }

    @PreDelete
    public void logMagazineDeletion ()
    {
        getLog ().debug ("deleting magazine containing" + photos.size ()
            + " photos.");
    }
}
```

In an XML mapping file, we can define the same methods without annotations:

```
<entity class="Magazine">
  <pre-remove>logMagazineDeletion</pre-remove>
  <post-load>convertPhotos</post-load>
</entity>
```

Note

We fully explore persistence metadata annotations and XML in [Chapter 5, Metadata \[28\]](#)

4.3.3. Using Entity Listeners

Mixing lifecycle event code into your persistent classes is not always ideal. It is often more elegant to handle cross-cutting lifecycle events in a non-persistent listener class. JPA allows for this, requiring only that listener classes have a public no-arg constructor. Like persistent classes, your listener classes can consume any number of callbacks. The callback methods must take in a single `java.lang.Object` argument which represents the persistent object that triggered the event.

Entities can enumerate listeners using the `EntityListeners` annotation. This annotation takes an array of listener classes as its value.

Below is an example of how to declare an entity and its corresponding listener classes.

```
/**
 * Example persistent class declaring our entity listener.
 */
@Entity
@EntityListeners({ MagazineLogger.class, ... })
public class Magazine
{
    // ... //
}

/**
 * Example entity listener.
 */
public class MagazineLogger
{
    @PostPersist
    public void logAddition (Object pc)
    {
        getLog ().debug ("Added new magazine:" + ((Magazine) pc).getTitle ());
    }

    @PreRemove
    public void logDeletion (Object pc)
    {
        getLog ().debug ("Removing from circulation:" +
            ((Magazine) pc).getTitle ());
    }
}
```

In XML we define both the listeners and their callback methods this way:

```
<entity class="Magazine">
  <entity-listeners>
    <entity-listener class="MagazineLogger">
      <post-persist>logAddition</post-persist>
      <pre-remove>logDeletion</pre-remove>
    </entity-listener>
  </entity-listeners>
</entity>
```

4.3.4. Entity Listeners Hierarchy

Entity listener methods are invoked in a specific order when a given event is fired. So-called *default* listeners are invoked first: these are listeners which have been defined in a package annotation or in the root element of XML mapping files. Next, entity listeners are invoked in the order of the inheritance hierarchy, with superclass listeners being invoked before subclass listeners. Finally, if an entity has multiple listeners for the same event, the listeners are invoked in declaration order.

You can exclude default listeners, and listeners defined in superclasses, from the invocation chain via two class-level annotations:

- `ExcludeDefaultListeners`: This annotation indicates that no default listeners will be invoked for this class, or any of its subclasses. The XML equivalent is the empty `exclude-default-listeners` element.
- `ExcludeSuperclassListeners`: This annotation will cause Kodo to skip invoking any listeners declared in superclasses. The XML equivalent is empty the `exclude-superclass-listeners` element.

4.4. Conclusions

This chapter covered everything you need to know to write persistent class definitions in JPA. JPA cannot use your persistent classes, however, until you complete one additional step: you must define the persistence metadata. The next chapter explores metadata in detail.

Chapter 5. Metadata

JPA requires that you accompany each persistent class with persistence metadata. This metadata serves three primary purposes:

1. To identify persistent classes.
2. To override default JPA behavior.
3. To provide the JPA implementation with information that it cannot glean from simply reflecting on the persistent class.

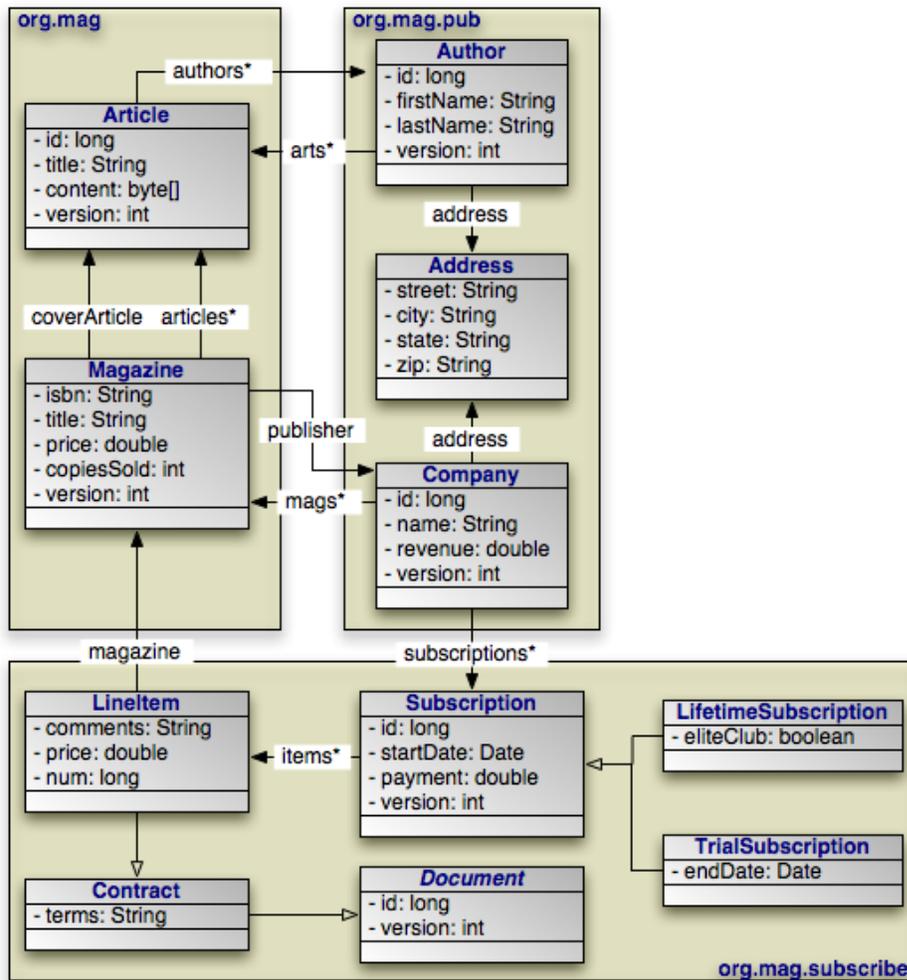
Persistence metadata is specified using either the Java 5 annotations defined in the `javax.persistence` package, XML mapping files, or a mixture of both. In the latter case, XML declarations override conflicting annotations. If you choose to use XML metadata, the XML files must be available at development and runtime, and must be discoverable via either of two strategies:

1. In a resource named `orm.xml` placed in the `META-INF` directory of the classpath or the jar archive containing your persistent classes.
2. As declared in your `persistence.xml` configuration file. In this case, each XML metadata file must be listed in a `mapping-file` element whose content is either a path to the given file or a resource location available to the class loader of the class.

We describe the standard metadata annotations and XML equivalents throughout this chapter. The full schema for XML mapping files is available in [Section 5.3, “XML Schema” \[42\]](#) JPA also standardizes relational mapping metadata and named query metadata, which we discuss in [Chapter 12, *Mapping Metadata* \[118\]](#) and [Section 10.1.9, “Named Queries” \[90\]](#) respectively.

Note

Kodo defines many useful annotations beyond the standard set. See [Section 6.3, “Additional JPA Metadata” \[481\]](#) and [Section 6.4, “Metadata Extensions” \[483\]](#) in the Reference Guide for details. There are currently no XML equivalents for these extension annotations.



Through the course of this chapter, we will create the persistent object model above.

5.1. Class Metadata

The following metadata annotations and XML elements apply to persistent class declarations.

5.1.1. Entity

The `Entity` annotation denotes an entity class. All entity classes must have this annotation. The `Entity` annotation takes one optional property:

- `String name`: Name used to refer to the entity in queries. It must not be a reserved literal in JPQL. It defaults to the unqualified name of the entity class.

The equivalent XML element is `entity`. It has the following attributes:

- `class`: The entity class. This attribute is required.

- `name`: Named used to refer to the class in queries. See the `name` property above.
- `access`: The access type to use for the class. It must either be `FIELD` or `PROPERTY`. For details on access types, see [Section 5.2, “Field and Property Metadata” \[33\]](#)

Note

Kodo uses a process called *enhancement* to modify the bytecode of entities for transparent lazy loading and immediate dirty tracking. See [Section 5.2, “Enhancement” \[452\]](#) in the Reference Guide for details on enhancement.

5.1.2. Id Class

As we discussed in [Section 4.2.1, “Identity Class” \[22\]](#), entities with multiple identity fields must use an *identity class* to encapsulate their persistent identity. The `IdClass` annotation specifies this class. It accepts a single `java.lang.Class` value.

The equivalent XML element is `id-class`, which has a single attribute:

- `class`: This required attribute lists the class name for the identity class.

5.1.3. Mapped Superclass

A *mapped superclass* is a non-entity class that can define persistent state and mapping information for entity subclasses. Mapped superclasses are usually abstract. Unlike true entities, you cannot query a mapped superclass, pass a mapped superclass instance to any `EntityManager` or `Query` methods, or declare a persistent relation with a mapped superclass target. You denote a mapped superclass with the `MappedSuperclass` marker annotation.

The equivalent XML element is `mapped-superclass`. It expects the following attributes:

- `class`: The entity class. This attribute is required.
- `access`: The access type to use for the class. It must either be `FIELD` or `PROPERTY`. For details on access types, see [Section 5.2, “Field and Property Metadata” \[33\]](#)

Note

Kodo allows you to query on mapped superclasses. A query on a mapped superclass will return all matching subclass instances. Kodo also allows you to declare relations to mapped superclass types; however, you cannot query across these relations.

5.1.4. Embeddable

The `Embeddable` annotation designates an embeddable persistent class. Embeddable instances are stored as part of the record of their owning instance. All embeddable classes must have this annotation.

A persistent class can either be an entity or an embeddable class, but not both.

The equivalent XML element is `embeddable`. It understands the following attributes:

- `class`: The entity class. This attribute is required.

- `access`: The access type to use for the class. It must either be `FIELD` or `PROPERTY`. For details on access types, see [Section 5.2, “Field and Property Metadata” \[33\]](#)

Note

Kodo allows a persistent class to be both an entity and an embeddable class. Instances of the class will act as entites when persisted explicitly or assigned to non-embedded fields of entities. Instances will act as embedded values when assigned to embedded fields of entities.

To signal that a class is both an entity and an embeddable class in Kodo, simply add both the `@Entity` and the `@Embeddable` annotations to the class.

5.1.5. EntityListeners

An entity may list its lifecycle event listeners in the `EntityListeners` annotation. This value of this annotation is an array of the listener `Classes` for the entity. The equivalent XML element is `entity-listeners`. For more details on entity listeners, see [Section 4.3, “Lifecycle Callbacks” \[24\]](#).

5.1.6. Example

Here are the class declarations for our persistent object model, annotated with the appropriate persistence metadata. Note that `Magazine` declares an identity class, and that `Document` and `Address` are a mapped superclass and an embeddable class, respectively. `LifetimeSubscription` and `TrialSubscription` override the default entity name to supply a shorter alias for use in queries.

Example 5.1. Class Metadata

```
package org.mag;

@Entity
@IdClass(Magazine.MagazineId.class)
public class Magazine
{
    ...

    public static class MagazineId
    {
        ...
    }
}

@Entity
public class Article
{
    ...
}

package org.mag.pub;

@Entity
public class Company
{
    ...
}

@Entity
public class Author
{
    ...
}

@Embeddable
public class Address
{
    ...
}
```

```

package org.mag.subscribe;

@MappedSuperclass
public abstract class Document
{
    ...
}

@Entity
public class Contract
    extends Document
{
    ...
}

@Entity
public class Subscription
{
    ...

    @Entity
    public static class LineItem
        extends Contract
    {
        ...
    }
}

@Entity(name="Lifetime")
public class LifetimeSubscription
    extends Subscription
{
    ...
}

@Entity(name="Trial")
public class TrialSubscription
    extends Subscription
{
    ...
}

```

The equivalent declarations in XML:

```

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
    version="1.0">
    <mapped-superclass class="org.mag.subscribe.Document">
        ...
    </mapped-superclass>
    <entity class="org.mag.Magazine">
        <id-class class="org.mag.Magazine$MagazineId"/>
        ...
    </entity>
    <entity class="org.mag.Article">
        ...
    </entity>
    <entity class="org.mag.pub.Company">
        ...
    </entity>
    <entity class="org.mag.pub.Author">
        ...
    </entity>
    <entity class="org.mag.subscribe.Contract">
        ...
    </entity>
    <entity class="org.mag.subscribe.LineItem">
        ...
    </entity>
    <entity class="org.mag.subscribe.LifetimeSubscription" name="Lifetime">
        ...
    </entity>
    <entity class="org.mag.subscribe.TrialSubscription" name="Trial">
        ...
    </entity>
    <embeddable class="org.mag.pub.Address">
        ...
    </embeddable>
</entity-mappings>

```

5.2. Field and Property Metadata

The persistence implementation must be able to retrieve and set the persistent state of your entities, mapped superclasses, and embeddable types. JPA offers two modes of persistent state access: *field access*, and *property access*. Under field access, the implementation injects state directly into your persistent fields, and retrieves changed state from your fields as well. To declare field access on an entity with XML metadata, set the `access` attribute of your `entity` XML element to `FIELD`. To use field access for an entity using annotation metadata, simply place your metadata and mapping annotations on your field declarations:

```
@ManyToOne
private Company publisher;
```

Property access, on the other hand, retrieves and loads state through JavaBean "getter" and "setter" methods. For a property `p` of type `T`, you must define the following getter method:

```
T getP ();
```

For boolean properties, this is also acceptable:

```
boolean isP ();
```

You must also define the following setter method:

```
void setP (T value);
```

To use property access, set your `entity` element's `access` attribute to `PROPERTY`, or place your metadata and mapping annotations on the getter method:

```
@ManyToOne
private Company getPublisher () { ... }

private void setPublisher (Company publisher) { ... }
```

Warning

When using property access, only the getter and setter method for a property should ever access the underlying persistent field directly. Other methods, including internal business methods in the persistent class, should go through the getter and setter methods when manipulating persistent state.

Also, take care when adding business logic to your getter and setter methods. Consider that they are invoked by the persistence implementation to load and retrieve all persistent state; other side effects might not be desirable.

Each class must use either field access or property access for all state; you cannot use both access types within the same class.

Additionally, a subclass must use the same access type as its superclass.

The remainder of this document uses the term "persistent field" to refer to either a persistent field or a persistent property.

5.2.1. Transient

The `Transient` annotation specifies that a field is non-persistent. Use it to exclude fields from management that would otherwise be persistent. `Transient` is a marker annotation only; it has no properties.

The equivalent XML element is `transient`. It has a single attribute:

- `name`: The transient field or property name. This attribute is required.

5.2.2. Id

Annotate your simple identity fields with `Id`. This annotation has no properties. We explore entity identity and identity fields in [Section 4.1.3, “Identity Fields” \[19\]](#).

The equivalent XML element is `id`. It has one required attribute:

- `name`: The name of the identity field or property.

5.2.3. Generated Value

The previous section showed you how to declare your identity fields with the `Id` annotation. It is often convenient to allow the persistence implementation to assign a unique value to your identity fields automatically. JPA includes the `GeneratedValue` annotation for this purpose. You can only apply the `GeneratedValue` annotation to numeric fields. It has the following properties:

- `GenerationType strategy`: Enum value specifying how to auto-generate the field value. The `GenerationType` enum has the following values:
 - `GeneratorType.AUTO`: The default. Assign the field a generated value, leaving the details to the JPA vendor.
 - `GenerationType.IDENTITY`: The database will assign an identity value on insert.
 - `GenerationType.SEQUENCE`: Use a datastore sequence to generate a field value.
 - `GenerationType.TABLE`: Use a sequence table to generate a field value.
- `String generator`: The name of a generator defined in mapping metadata. We show you how to define named generators in [Section 12.5, “Generators” \[126\]](#) If the `GenerationType` is set but this property is unset, the JPA implementation uses appropriate defaults for the selected generation type.

The equivalent XML element is `generated-value`, which includes the following attributes:

- `strategy`: One of `TABLE`, `SEQUENCE`, `IDENTITY`, or `AUTO`, defaulting to `AUTO`.
- `generator`: Equivalent to the generator property listed above.

Note

Kodo allows you to use the `GeneratedValue` annotation on any field, not just identity fields. Before using the `IDENTITY` generation strategy, however, read [Section 5.3.3, “Autoassign / Identity Strategy Caveats” \[458\]](#) in the Reference Guide.

Kodo also offers two additional generator strategies for non-numeric fields, which you can access by setting `strategy` to `AUTO` (the default), and setting the `generator` string to:

- `uuid-string`: Kodo will generate a 128-bit UUID unique within the network, represented as a 16-character string. For more information on UUIDs, see the IETF UUID draft specification at: <http://www1.ics.uci.edu/~ejw/authoring/uuid-guid/>
- `uuid-hex`: Same as `uuid-string`, but represents the UUID as a 32-character hexadecimal string.

These string constants are defined in `kodo.persistence.Generator`.

5.2.4. Embedded Id

If your entity has multiple identity values, you may declare multiple `@Id` fields, or you may declare a single `@EmbeddedId` field. The type of a field annotated with `EmbeddedId` must be an embeddable entity class. The fields of this embeddable class are considered the identity values of the owning entity. We explore entity identity and identity fields in [Section 4.1.3, “Identity Fields” \[19\]](#).

The `EmbeddedId` annotation has no properties.

The equivalent XML element is `embedded-id`. It has one required attribute:

- `name`: The name of the identity field or property.

5.2.5. Version

Use the `Version` annotation to designate a version field. [Section 4.1.4, “Version Field” \[19\]](#) explained the importance of version fields to JPA. This is a marker annotation; it has no properties.

The equivalent XML element is `version`, which has a single attribute:

- `name`: The name of the version field or property. This attribute is required.

5.2.6. Basic

`Basic` signifies a standard value persisted as-is to the datastore. You can use the `Basic` annotation on persistent fields of the following types: primitives, primitive wrappers, `java.lang.String`, `byte[]`, `Byte[]`, `char[]`, `Character[]`, `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Timestamp`, `Enums`, and `Serializable` types.

`Basic` declares these properties:

- `FetchType fetch`: Whether to load the field eagerly (`FetchType.EAGER`) or lazily (`FetchType.LAZY`). The de-

fault is `FetchType.EAGER`.

- `boolean optional`: Whether the datastore allows null values. The default is true.

The equivalent XML element is `basic`. It has the following attributes:

- `name`: The name of the field or property. This attribute is required.
- `fetch`: One of `EAGER` or `LAZY`.
- `optional`: Boolean indicating whether the field value may be null.

5.2.6.1. Fetch Type

Many metadata annotations in JPA have a `fetch` property. This property can take on one of two values: `FetchType.EAGER` or `FetchType.LAZY`. `FetchType.EAGER` means that the field is loaded by the JPA implementation before it returns the persistent object to you. Whenever you retrieve an entity from a query or from the `EntityManager`, you are guaranteed that all of its eager fields are populated with datastore data.

`FetchType.LAZY` is a hint to the JPA runtime that you want to defer loading of the field until you access it. This is called *lazy loading*. Lazy loading is completely transparent; when you attempt to read the field for the first time, the JPA runtime will load the value from the datastore and populate the field automatically. Lazy loading is only a hint and not a directive because some JPA implementations cannot lazy-load certain field types.

With a mix of eager and lazily-loaded fields, you can ensure that commonly-used fields load efficiently, and that other state loads transparently when accessed. As you will see in [Section 7.3, “Persistence Context” \[67\]](#) you can also use eager fetching to ensure that entities have all needed data loaded before they become *detached* at the end of a persistence context.

Note

Kodo can lazy-load any field type. Kodo also allows you to dynamically change which fields are eagerly or lazily loaded at runtime. See [Section 5.6, “Fetch Groups” \[469\]](#) in the Reference Guide for details.

The Reference Guide details Kodo's eager fetching behavior in [Section 5.7, “Eager Fetching” \[473\]](#)

5.2.7. Embedded

Use the `Embedded` marker annotation on embeddable field types. Embedded fields are mapped as part of the datastore record of the declaring entity. In our sample model, `Author` and `Company` each embed their `Address`, rather than forming a relation to an `Address` as a separate entity.

The equivalent XML element is `embedded`, which expects a single attribute:

- `name`: The name of the field or property. This attribute is required.

5.2.8. Many-To-One

When an entity A references a single entity B, and other As might also reference the same B, we say there is a *many-to-one* relation from A to B. In our sample model, for example, each magazine has a reference to its publisher. Multiple magazines might have the same publisher. We say, then, that the `Magazine.publisher` field is a many-to-one relation from magazines to publishers.

JPA indicates many-to-one relations between entities with the `ManyToOne` annotation. This annotation has the following properties:

- `Class targetEntity`: The class of the related entity type.
- `CascadeType[] cascade`: Array of enum values defining cascade behavior for this field. We explore cascades below. The default is an empty array.
- `FetchType fetch`: Whether to load the field eagerly (`FetchType.EAGER`) or lazily (`FetchType.LAZY`). The default is `FetchType.EAGER`. See [Section 5.2.6.1, “Fetch Type” \[36\]](#) above for details on fetch types.
- `boolean optional`: Whether the related object must exist. If `false`, this field cannot be null. The default is `true`.

The equivalent XML element is `many-to-one`. It accepts the following attributes:

- `name`: The name of the field or property. This attribute is required.
- `target-entity`: The class of the related type.
- `fetch`: Either `EAGER` or `LAZY`.
- `optional`: Boolean indicating whether the field value may be null.

5.2.8.1. Cascade Type

We introduce the JPA `EntityManager` in [Chapter 8, *EntityManager* \[70\]](#). The `EntityManager` has APIs to persist new entities, remove (delete) existing entities, refresh entity state from the datastore, and merge *detached* entity state back into the persistence context. We explore all of these APIs in detail later in the overview.

When the `EntityManager` is performing the above operations, you can instruct it to automatically cascade an operation to the entities held in a persistent field with the `cascade` property of your metadata annotation. This process is recursive. The `cascade` property accepts an array of `CascadeType` enum values.

- `CascadeType.PERSIST`: When persisting an entity, also persist the entities held in this field. We suggest liberal application of this cascade rule, because if the `EntityManager` finds a field that references a new entity during flush, and the field does not use `CascadeType.PERSIST`, it is an error.
- `CascadeType.REMOVE`: When deleting an entity, also delete the entities held in this field.
- `CascadeType.REFRESH`: When refreshing an entity, also refresh the entities held in this field.
- `CascadeType.MERGE`: When merging entity state, also merge the entities held in this field.

`CascadeType` defines one additional value, `CascadeType.ALL`, that acts as a shortcut for all of the values above. The following annotations are equivalent:

```
@ManyToOne(cascade={CascadeType.PERSIST,CascadeType.REMOVE,
    CascadeType.REFRESH,CascadeType.MERGE})
private Company publisher;
```

```
@ManyToOne(cascade=CascadeType.ALL)
private Company publisher;
```

In XML, these enumeration constants are available as child elements of the `cascade` element. The `cascade` element is itself a child of `many-to-one`. The following examples are equivalent:

```
<many-to-one name="publisher">
  <cascade>
    <cascade-persist/>
    <cascade-merge/>
    <cascade-remove/>
    <cascade-refresh/>
  </cascade>
</many-to-one>
```

```
<many-to-one name="publisher">
  <cascade>
    <cascade-all/>
  </cascade>
</many-to-one>
```

5.2.9. One-To-Many

When an entity A references multiple B entities, and no two As reference the same B, we say there is a *one-to-many* relation from A to B.

One-to-many relations are the exact inverse of the many-to-one relations we detailed in the preceding section. In that section, we said that the `Magazine.publisher` field is a many-to-one relation from magazines to publishers. But now we see that the `Company.mags` field is the inverse - a one-to-many relation from publishers to magazines. Each company may publish multiple magazines, but each magazine can have only one publisher.

JPA indicates one-to-many relations between entities with the `OneToMany` annotation. This annotation has the following properties:

- `Class targetEntity`: The class of the related entity type. This information is usually taken from the parameterized collection or map element type. You must supply it explicitly, however, if your field isn't a parameterized type.
- `String mappedBy`: Names the many-to-one field in the related entity that maps this bidirectional relation. We explain bidirectional relations below. Leaving this property unset signals that this is a standard unidirectional relation.
- `CascadeType[] cascade`: Array of enum values defining cascade behavior for the collection elements. We explore cascades above in [Section 5.2.8.1, "Cascade Type" \[37\]](#). The default is an empty array.
- `FetchType fetch`: Whether to load the field eagerly (`FetchType.EAGER`) or lazily (`FetchType.LAZY`). The default is `FetchType.LAZY`. See [Section 5.2.6.1, "Fetch Type" \[36\]](#) above for details on fetch types.

The equivalent XML element is `one-to-many`, which includes the following attributes:

- `name`: The name of the field or property. This attribute is required.
- `target-entity`: The class of the related type.

- `fetch`: Either `EAGER` or `LAZY`.
- `mapped-by`: The name of the field or property that owns the relation. See [Section 5.2, “Field and Property Metadata” \[33\]](#).

You may also nest the `cascade` element within a `one-to-many` element.

5.2.9.1. Bidirectional Relations

When two fields are logical inverses of each other, they form a *bidirectional relation*. Our model contains two bidirectional relations: `Magazine.publisher` and `Company.mags` form one bidirectional relation, and `Article.authors` and `Author.articles` form the other. In both cases, there is a clear link between the two fields that form the relationship. A magazine refers to its publisher while the publisher refers to all its published magazines. An article refers to its authors while each author refers to his or her written articles.

When the two fields of a bidirectional relation share the same datastore mapping, JPA formalizes the connection with the `mappedBy` property. Marking `Company.mags` as `mappedBy Magazine.publisher` means two things:

1. `Company.mags` uses the datastore mapping for `Magazine.publisher`, but inverts it. In fact, it is illegal to specify any additional mapping information when you use the `mappedBy` property. All mapping information is read from the referenced field. We explore mapping in depth in [Chapter 12, Mapping Metadata \[118\]](#)
2. `Magazine.publisher` is the "owner" of the relation. The field that specifies the mapping data is always the owner. This means that changes to the `Magazine.publisher` field are reflected in the datastore, while changes to the `Company.mags` field alone are not. Changes to `Company.mags` may still affect the JPA implementation's cache, however. Thus, it is very important that you keep your object model consistent by properly maintaining both sides of your bidirectional relations at all times.

You should always take advantage of the `mappedBy` property rather than mapping each field of a bidirectional relation independently. Failing to do so may result in the JPA implementation trying to update the database with conflicting data. Be careful, however, to only mark one side of the relation as `mappedBy`. One side has to actually do the mapping!

Note

You can configure Kodo to automatically synchronize both sides of a bidirectional relation, or to perform various actions when it detects inconsistent relations. See [Section 5.4, “Managed Inverses” \[458\]](#) in the Reference Guide for details.

5.2.10. One-To-One

When an entity A references a single entity B, and no other As can reference the same B, we say there is a *one-to-one* relation between A and B. In our sample model, `Magazine` has a one-to-one relation to `Article` through the `Magazine.coverArticle` field. No two magazines can have the same cover article.

JPA indicates one-to-one relations between entities with the `OneToOne` annotation. This annotation has the following properties:

- `Class targetEntity`: The class of the related entity type. This information is usually taken from the field type.
- `String mappedBy`: Names the field in the related entity that maps this bidirectional relation. We explain bidirectional relations in [Section 5.2.9.1, “Bidirectional Relations” \[39\]](#) above. Leaving this property unset signals that this is a standard unidirectional relation.
- `CascadeType[] cascade`: Array of enum values defining cascade behavior for this field. We explore cascades in [Sec-](#)

tion 5.2.8.1, “Cascade Type” [37] above. Defaults to an empty array.

- `FetchType fetch`: Whether to load the field eagerly (`FetchType.EAGER`) or lazily (`FetchType.LAZY`). The default is `FetchType.EAGER`. See **Section 5.2.6.1, “Fetch Type” [36]** above for details on fetch types.
- `boolean optional`: Whether the related object must exist. If `false`, this field cannot be null. The default is `true`.

The equivalent XML element is `one-to-one`, which understands the following attributes:

- `name`: The name of the field or property. This attribute is required.
- `target-entity`: The class of the related type.
- `fetch`: Either `EAGER` or `LAZY`.
- `mapped-by`: The field that owns the relation. See **Section 5.2, “Field and Property Metadata” [33]**.

You may also nest the `cascade` element within a `one-to-one` element.

5.2.11. Many-To-Many

When an entity `A` references multiple `B` entities, and other `As` might reference some of the same `Bs`, we say there is a *many-to-many* relation between `A` and `B`. In our sample model, for example, each article has a reference to all the authors that contributed to the article. Other articles might have some of the same authors. We say, then, that `Article` and `Author` have a many-to-many relation through the `Article.authors` field.

JPA indicates many-to-many relations between entities with the `ManyToMany` annotation. This annotation has the following properties:

- `Class targetEntity`: The class of the related entity type. This information is usually taken from the parameterized collection or map element type. You must supply it explicitly, however, if your field isn't a parameterized type.
- `String mappedBy`: Names the many-to-many field in the related entity that maps this bidirectional relation. We explain bidirectional relations in **Section 5.2.9.1, “Bidirectional Relations” [39]** above. Leaving this property unset signals that this is a standard unidirectional relation.
- `CascadeType[] cascade`: Array of enum values defining cascade behavior for the collection elements. We explore cascades above in **Section 5.2.8.1, “Cascade Type” [37]**. The default is an empty array.
- `FetchType fetch`: Whether to load the field eagerly (`FetchType.EAGER`) or lazily (`FetchType.LAZY`). The default is `FetchType.LAZY`. See **Section 5.2.6.1, “Fetch Type” [36]** above for details on fetch types.

The equivalent XML element is `many-to-many`. It accepts the following attributes:

- `name`: The name of the field or property. This attribute is required.
- `target-entity`: The class of the related type.
- `fetch`: Either `EAGER` or `LAZY`.
- `mapped-by`: The field that owns the relation. See **Section 5.2, “Field and Property Metadata” [33]**.

You may also nest the `cascade` element within a `many-to-many` element.

5.2.12. Order By

Datastores such as relational databases do not preserve the order of records. Your persistent `List` fields might be ordered one way the first time you retrieve an object from the datastore, and a completely different way the next. To ensure consistent ordering of collection fields, you must use the `OrderBy` annotation. The `OrderBy` annotation's value is a string defining the order of the collection elements. An empty value means to sort on the identity value(s) of the elements in ascending order. Any other value must be of the form:

```
<field name>[ ASC|DESC][, ...]
```

Each `<field name>` is the name of a persistent field in the collection's element type. You can optionally follow each field by the keyword `ASC` for ascending order, or `DESC` for descending order. If the direction is omitted, it defaults to ascending.

The equivalent XML element is `order-by`, which can be listed as a sub-element of the `one-to-many` or `many-to-many` elements. The text within this element is parsed as the order by string.

Note

Kodo expands the available ordering syntax. See [Section 6.4.2.3, “Order-By” \[485\]](#) in the Reference Guide for details.

5.2.13. Map Key

JPA supports persistent `Map` fields through either a **OneToMany** or **ManyToMany** association. The related entities form the map values. JPA derives the map keys by extracting a field from each entity value. The `MapKey` annotation designates the field that is used as the key. It has the following properties:

- `String name`: The name of a field in the related entity class to use as the map key. If no name is given, the default is the identity field of the related entity class.

The equivalent XML element is `map-key`, which can be listed as a sub-element of the `one-to-many` or `many-to-many` elements. The `map-key` element has the following attributes:

- `name`: The name of the field in the related entity class to use as the map key.

5.2.14. Persistent Field Defaults

In the absence of any of the annotations above, JPA defines the following default behavior for declared fields:

1. Fields declared `static`, `transient`, or `final` default to non-persistent.
2. Fields of any primitive type, primitive wrapper type, `java.lang.String`, `byte[]`, `Byte[]`, `char[]`, `Character[]`, `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Timestamp`, or any `Serializable` type default to persistent, as if annotated with `@Basic`.
3. Fields of an embeddable type default to persistent, as if annotated with `@Embedded`.
4. All other fields default to non-persistent.

Note that according to these defaults, all relations between entities must be annotated explicitly. Without an annotation, a relation field will default to serialized storage if the related entity type is serializable, or will default to being non-persistent if not.

5.3. XML Schema

We present the complete XML schema below. Many of the elements relate to object/relational mapping rather than metadata; these elements are discussed in [Chapter 12, Mapping Metadata \[118\]](#)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:orm="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="1.0">

  <xsd:annotation>
    <xsd:documentation>
      @(#)orm_1_0.xsd 1.0 Feb 14 2006
    </xsd:documentation>
  </xsd:annotation>
  <xsd:annotation>
    <xsd:documentation>

      This is the XML Schema for the persistence object-relational
      mapping file.
      The file may be named "META-INF/orm.xml" in the persistence
      archive or it may be named some other name which would be
      used to locate the file as resource on the classpath.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:complexType name="emptyType"/>

  <xsd:simpleType name="versionType">
    <xsd:restriction base="xsd:token">
      <xsd:pattern value="[0-9]+(\.[0-9]+)*"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- ***** -->

  <xsd:element name="entity-mappings">
    <xsd:complexType>
      <xsd:annotation>
        <xsd:documentation>

          The entity-mappings element is the root element of an mapping
          file. It contains the following four types of elements:

          1. The persistence-unit-metadata element contains metadata
          for the entire persistence unit. It is undefined if this element
          occurs in multiple mapping files within the same persistence unit.

          2. The package, schema, catalog and access elements apply to all of
          the entity, mapped-superclass and embeddable elements defined in
          the same file in which they occur.

          3. The sequence-generator, table-generator, named-query,
          named-native-query and sql-result-set-mapping elements are global
          to the persistence unit. It is undefined to have more than one
          sequence-generator or table-generator of the same name in the same
          or different mapping files in a persistence unit. It is also
          undefined to have more than one named-query or named-native-query
          of the same name in the same or different mapping files in a
          persistence unit.

          4. The entity, mapped-superclass and embeddable elements each define
          the mapping information for a managed persistent class. The mapping
          information contained in these elements may be complete or it may
          be partial.

        </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
        <xsd:element name="description" type="xsd:string"
          minOccurs="0"/>
        <xsd:element name="persistence-unit-metadata"
          type="orm:persistence-unit-metadata"
          minOccurs="0"/>
        <xsd:element name="package" type="xsd:string"
          minOccurs="0"/>
        <xsd:element name="schema" type="xsd:string"
          minOccurs="0"/>
        <xsd:element name="catalog" type="xsd:string"

```

```

        minOccurs="0" />
<xsd:element name="access" type="orm:access-type"
  minOccurs="0" />
<xsd:element name="sequence-generator" type="orm:sequence-generator"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="table-generator" type="orm:table-generator"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="named-query" type="orm:named-query"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="named-native-query" type="orm:named-native-query"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="sql-result-set-mapping"
  type="orm:sql-result-set-mapping"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="mapped-superclass" type="orm:mapped-superclass"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="entity" type="orm:entity"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="embeddable" type="orm:embeddable"
  minOccurs="0" maxOccurs="unbounded" />
</xsd:sequence>
<xsd:attribute name="version" type="orm:versionType"
  fixed="1.0" use="required" />
</xsd:complexType>
</xsd:element>
<!-- ***** -->
<xsd:complexType name="persistence-unit-metadata">
  <xsd:annotation>
    <xsd:documentation>
      Metadata that applies to the persistence unit and not just to
      the mapping file in which it is contained.
      If the xml-mapping-metadata-complete element is specified then
      the complete set of mapping metadata for the persistence unit
      is contained in the XML mapping files for the persistence unit.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="xml-mapping-metadata-complete" type="orm:emptyType"
      minOccurs="0" />
    <xsd:element name="persistence-unit-defaults"
      type="orm:persistence-unit-defaults"
      minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="persistence-unit-defaults">
  <xsd:annotation>
    <xsd:documentation>
      These defaults are applied to the persistence unit as a whole
      unless they are overridden by local annotation or XML
      element settings.
      schema - Used as the schema for all tables or secondary tables
      that apply to the persistence unit
      catalog - Used as the catalog for all tables or secondary tables
      that apply to the persistence unit
      access - Used as the access type for all managed classes in
      the persistence unit
      cascade-persist - Adds cascade-persist to the set of cascade options
      in entity relationships of the persistence unit
      entity-listeners - List of default entity listeners to be invoked
      on each entity in the persistence unit.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="schema" type="xsd:string"
      minOccurs="0" />
    <xsd:element name="catalog" type="xsd:string"
      minOccurs="0" />
    <xsd:element name="access" type="orm:access-type"
      minOccurs="0" />
    <xsd:element name="cascade-persist" type="orm:emptyType"
      minOccurs="0" />
    <xsd:element name="entity-listeners" type="orm:entity-listeners"
      minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="entity">
  <xsd:annotation>
    <xsd:documentation>

```

```

Defines the settings and mappings for an entity. Is allowed to be
sparsely populated and used in conjunction with the annotations.
Alternatively, the metadata-complete attribute can be used to
indicate that no annotations on the entity class (and its fields
or properties) are to be processed. If this is the case then
the defaulting rules for the entity and its subelements will
be recursively applied.

@Target(TYPE) @Retention(RUNTIME)
public @interface Entity {
    String name() default "";
}

</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
  <xsd:element name="description" type="xsd:string" minOccurs="0"/>
  <xsd:element name="table" type="orm:table"
    minOccurs="0"/>
  <xsd:element name="secondary-table" type="orm:secondary-table"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="primary-key-join-column"
    type="orm:primary-key-join-column"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="id-class" type="orm:id-class" minOccurs="0"/>
  <xsd:element name="inheritance" type="orm:inheritance" minOccurs="0"/>
  <xsd:element name="discriminator-value" type="orm:discriminator-value"
    minOccurs="0"/>
  <xsd:element name="discriminator-column"
    type="orm:discriminator-column"
    minOccurs="0"/>
  <xsd:element name="sequence-generator" type="orm:sequence-generator"
    minOccurs="0"/>
  <xsd:element name="table-generator" type="orm:table-generator"
    minOccurs="0"/>
  <xsd:element name="named-query" type="orm:named-query"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="named-native-query" type="orm:named-native-query"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="sql-result-set-mapping"
    type="orm:sql-result-set-mapping"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="exclude-default-listeners" type="orm:emptyType"
    minOccurs="0"/>
  <xsd:element name="exclude-superclass-listeners" type="orm:emptyType"
    minOccurs="0"/>
  <xsd:element name="entity-listeners" type="orm:entity-listeners"
    minOccurs="0"/>
  <xsd:element name="pre-persist" type="orm:pre-persist" minOccurs="0"/>
  <xsd:element name="post-persist" type="orm:post-persist"
    minOccurs="0"/>
  <xsd:element name="pre-remove" type="orm:pre-remove" minOccurs="0"/>
  <xsd:element name="post-remove" type="orm:post-remove" minOccurs="0"/>
  <xsd:element name="pre-update" type="orm:pre-update" minOccurs="0"/>
  <xsd:element name="post-update" type="orm:post-update" minOccurs="0"/>
  <xsd:element name="post-load" type="orm:post-load" minOccurs="0"/>
  <xsd:element name="attribute-override" type="orm:attribute-override"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="association-override"
    type="orm:association-override"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="attributes" type="orm:attributes" minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="name" type="xsd:string"/>
<xsd:attribute name="class" type="xsd:string" use="required"/>
<xsd:attribute name="access" type="orm:access-type"/>
<xsd:attribute name="metadata-complete" type="xsd:boolean"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="attributes">
  <xsd:annotation>
    <xsd:documentation>

      This element contains the entity field or property mappings.
      It may be sparsely populated to include only a subset of the
      fields or properties. If metadata-complete for the entity is true
      then the remainder of the attributes will be defaulted according
      to the default rules.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="id" type="orm:id"
        minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="embedded-id" type="orm:embedded-id"
        minOccurs="0"/>
    </xsd:choice>
    <xsd:element name="basic" type="orm:basic"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="version" type="orm:version"
  </xsd:complexType>

```

```

        minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="many-to-one" type="orm:many-to-one"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="one-to-many" type="orm:one-to-many"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="one-to-one" type="orm:one-to-one"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="many-to-many" type="orm:many-to-many"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="embedded" type="orm:embedded"
  minOccurs="0" maxOccurs="unbounded" />
<xsd:element name="transient" type="orm:transient"
  minOccurs="0" maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>

<!-- ***** -->

<xsd:simpleType name="access-type">
  <xsd:annotation>
    <xsd:documentation>

      This element determines how the persistence provider accesses the
      state of an entity or embedded object.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="PROPERTY" />
    <xsd:enumeration value="FIELD" />
  </xsd:restriction>
</xsd:simpleType>

<!-- ***** -->

<xsd:complexType name="entity-listeners">
  <xsd:annotation>
    <xsd:documentation>

      @Target({TYPE}) @Retention(RUNTIME)
      public @interface EntityListeners {
        Class[] value();
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="entity-listener" type="orm:entity-listener"
      minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="entity-listener">
  <xsd:annotation>
    <xsd:documentation>

      Defines an entity listener to be invoked at lifecycle events
      for the entities that list this listener.

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="pre-persist" type="orm:pre-persist" minOccurs="0" />
    <xsd:element name="post-persist" type="orm:post-persist"
      minOccurs="0" />
    <xsd:element name="pre-remove" type="orm:pre-remove" minOccurs="0" />
    <xsd:element name="post-remove" type="orm:post-remove" minOccurs="0" />
    <xsd:element name="pre-update" type="orm:pre-update" minOccurs="0" />
    <xsd:element name="post-update" type="orm:post-update" minOccurs="0" />
    <xsd:element name="post-load" type="orm:post-load" minOccurs="0" />
  </xsd:sequence>
  <xsd:attribute name="class" type="xsd:string" use="required" />
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="pre-persist">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PrePersist {}

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="method-name" type="xsd:string" use="required" />
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="post-persist">

```

```
<xsd:annotation>
  <xsd:documentation>

    @Target({METHOD}) @Retention(RUNTIME)
    public @interface PostPersist {}

  </xsd:documentation>
</xsd:annotation>
<xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="pre-remove">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreRemove {}

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="post-remove">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostRemove {}

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="pre-update">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PreUpdate {}

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="post-update">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostUpdate {}

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="post-load">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD}) @Retention(RUNTIME)
      public @interface PostLoad {}

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="query-hint">
  <xsd:annotation>
    <xsd:documentation>

      @Target({}) @Retention(RUNTIME)
      public @interface QueryHint {
        String name();
        String value();
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="method-name" type="xsd:string" use="required"/>
</xsd:complexType>
```

```

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="value" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="named-query">
  <xsd:annotation>
    <xsd:documentation>

      @Target({TYPE}) @Retention(RUNTIME)
      public @interface NamedQuery {
        String name();
        String query();
        QueryHint[] hints() default {};
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="query" type="xsd:string"/>
    <xsd:element name="hint" type="orm:query-hint"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="named-native-query">
  <xsd:annotation>
    <xsd:documentation>

      @Target({TYPE}) @Retention(RUNTIME)
      public @interface NamedNativeQuery {
        String name();
        String query();
        QueryHint[] hints() default {};
        Class resultClass() default void.class;
        String resultSetMapping() default ""; //named SqlResultSetMapping
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="query" type="xsd:string"/>
    <xsd:element name="hint" type="orm:query-hint"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="result-class" type="xsd:string"/>
  <xsd:attribute name="result-set-mapping" type="xsd:string"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="sql-result-set-mapping">
  <xsd:annotation>
    <xsd:documentation>

      @Target({TYPE}) @Retention(RUNTIME)
      public @interface SqlResultSetMapping {
        String name();
        EntityResult[] entities() default {};
        ColumnResult[] columns() default {};
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="entity-result" type="orm:entity-result"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="column-result" type="orm:column-result"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="entity-result">
  <xsd:annotation>
    <xsd:documentation>

      @Target({}) @Retention(RUNTIME)
      public @interface EntityResult {
        Class entityClass();
        FieldResult[] fields() default {};
        String discriminatorColumn() default "";
      }

    </xsd:documentation>
  </xsd:annotation>

```

```

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="field-result" type="orm:field-result"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="entity-class" type="xsd:string" use="required"/>
  <xsd:attribute name="discriminator-column" type="xsd:string"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="field-result">
  <xsd:annotation>
    <xsd:documentation>

      @Target({}) @Retention(RUNTIME)
      public @interface FieldResult {
        String name();
        String column();
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="column" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="column-result">
  <xsd:annotation>
    <xsd:documentation>

      @Target({}) @Retention(RUNTIME)
      public @interface ColumnResult {
        String name();
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="table">
  <xsd:annotation>
    <xsd:documentation>

      @Target({TYPE}) @Retention(RUNTIME)
      public @interface Table {
        String name() default "";
        String catalog() default "";
        String schema() default "";
        UniqueConstraint[] uniqueConstraints() default {};
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="unique-constraint" type="orm:unique-constraint"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="catalog" type="xsd:string"/>
  <xsd:attribute name="schema" type="xsd:string"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="secondary-table">
  <xsd:annotation>
    <xsd:documentation>

      @Target({TYPE}) @Retention(RUNTIME)
      public @interface SecondaryTable {
        String name();
        String catalog() default "";
        String schema() default "";
        PrimaryKeyJoinColumn[] pkJoinColumns() default {};
        UniqueConstraint[] uniqueConstraints() default {};
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="primary-key-join-column"
      type="orm:primary-key-join-column"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="unique-constraint" type="orm:unique-constraint"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>

```

```

<xsd:attribute name="name" type="xsd:string" use="required"/>
<xsd:attribute name="catalog" type="xsd:string"/>
<xsd:attribute name="schema" type="xsd:string"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="unique-constraint">
  <xsd:annotation>
    <xsd:documentation>

      @Target({}) @Retention(RUNTIME)
      public @interface UniqueConstraint {
        String[] columnNames();
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="column-name" type="xsd:string"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="column">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Column {
        String name() default "";
        boolean unique() default false;
        boolean nullable() default true;
        boolean insertable() default true;
        boolean updatable() default true;
        String columnDefinition() default "";
        String table() default "";
        int length() default 255;
        int precision() default 0; // decimal precision
        int scale() default 0; // decimal scale
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="unique" type="xsd:boolean"/>
  <xsd:attribute name="nullable" type="xsd:boolean"/>
  <xsd:attribute name="insertable" type="xsd:boolean"/>
  <xsd:attribute name="updatable" type="xsd:boolean"/>
  <xsd:attribute name="column-definition" type="xsd:string"/>
  <xsd:attribute name="table" type="xsd:string"/>
  <xsd:attribute name="length" type="xsd:int"/>
  <xsd:attribute name="precision" type="xsd:int"/>
  <xsd:attribute name="scale" type="xsd:int"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="join-column">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface JoinColumn {
        String name() default "";
        String referencedColumnName() default "";
        boolean unique() default false;
        boolean nullable() default true;
        boolean insertable() default true;
        boolean updatable() default true;
        String columnDefinition() default "";
        String table() default "";
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="referenced-column-name" type="xsd:string"/>
  <xsd:attribute name="unique" type="xsd:boolean"/>
  <xsd:attribute name="nullable" type="xsd:boolean"/>
  <xsd:attribute name="insertable" type="xsd:boolean"/>
  <xsd:attribute name="updatable" type="xsd:boolean"/>
  <xsd:attribute name="column-definition" type="xsd:string"/>
  <xsd:attribute name="table" type="xsd:string"/>
</xsd:complexType>

<!-- ***** -->

<xsd:simpleType name="generation-type">
  <xsd:annotation>
    <xsd:documentation>

```

```

    public enum GenerationType { TABLE, SEQUENCE, IDENTITY, AUTO };
</xsd:documentation>
</xsd:annotation>
<xsd:restriction base="xsd:token">
  <xsd:enumeration value="TABLE"/>
  <xsd:enumeration value="SEQUENCE"/>
  <xsd:enumeration value="IDENTITY"/>
  <xsd:enumeration value="AUTO"/>
</xsd:restriction>
</xsd:simpleType>
<!-- ***** -->
<xsd:complexType name="attribute-override">
  <xsd:annotation>
    <xsd:documentation>
      @Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
      public @interface AttributeOverride {
        String name();
        Column column();
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="column" type="orm:column"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="association-override">
  <xsd:annotation>
    <xsd:documentation>
      @Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
      public @interface AssociationOverride {
        String name();
        JoinColumn[] joinColumns();
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="join-column" type="orm:join-column"
      maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="id-class">
  <xsd:annotation>
    <xsd:documentation>
      @Target({TYPE}) @Retention(RUNTIME)
      public @interface IdClass {
        Class value();
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="class" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="id">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Id {}
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="column" type="orm:column"
      minOccurs="0"/>
    <xsd:element name="generated-value" type="orm:generated-value"
      minOccurs="0"/>
    <xsd:element name="temporal" type="orm:temporal"
      minOccurs="0"/>
    <xsd:element name="table-generator" type="orm:table-generator"
      minOccurs="0"/>
    <xsd:element name="sequence-generator" type="orm:sequence-generator"
      minOccurs="0"/>
  </xsd:sequence>

```

```

<xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="embedded-id">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface EmbeddedId {}
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="attribute-override" type="orm:attribute-override"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="transient">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Transient {}
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="version">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Version {}
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="column" type="orm:column" minOccurs="0"/>
    <xsd:element name="temporal" type="orm:temporal" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="basic">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Basic {
        FetchType fetch() default EAGER;
        boolean optional() default true;
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="column" type="orm:column" minOccurs="0"/>
    <xsd:choice>
      <xsd:element name="lob" type="orm:lob" minOccurs="0"/>
      <xsd:element name="temporal" type="orm:temporal" minOccurs="0"/>
      <xsd:element name="enumerated" type="orm:enumerated" minOccurs="0"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="fetch" type="orm:fetch-type"/>
  <xsd:attribute name="optional" type="xsd:boolean"/>
</xsd:complexType>
<!-- ***** -->
<xsd:simpleType name="fetch-type">
  <xsd:annotation>
    <xsd:documentation>
      public enum FetchType { LAZY, EAGER };
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="LAZY"/>
    <xsd:enumeration value="EAGER"/>
  </xsd:restriction>
</xsd:simpleType>

```

```

</xsd:restriction>
</xsd:simpleType>

<!-- ***** -->

<xsd:complexType name="lob">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Lob {}

    </xsd:documentation>
  </xsd:annotation>
</xsd:complexType>

<!-- ***** -->

<xsd:simpleType name="temporal">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Temporal {
        TemporalType value();
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="orm:temporal-type"/>
</xsd:simpleType>

<!-- ***** -->

<xsd:simpleType name="temporal-type">
  <xsd:annotation>
    <xsd:documentation>

      public enum TemporalType {
        DATE, // java.sql.Date
        TIME, // java.sql.Time
        TIMESTAMP // java.sql.Timestamp
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="DATE"/>
    <xsd:enumeration value="TIME"/>
    <xsd:enumeration value="TIMESTAMP"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- ***** -->

<xsd:simpleType name="enumerated">
  <xsd:annotation>
    <xsd:documentation>

      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Enumerated {
        EnumType value() default ORDINAL;
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="orm:enum-type"/>
</xsd:simpleType>

<!-- ***** -->

<xsd:simpleType name="enum-type">
  <xsd:annotation>
    <xsd:documentation>

      public enum EnumType {
        ORDINAL,
        STRING
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="ORDINAL"/>
    <xsd:enumeration value="STRING"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- ***** -->

<xsd:complexType name="many-to-one">
  <xsd:annotation>
    <xsd:documentation>

```

```

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
  <xsd:choice>
    <xsd:element name="join-column" type="orm:join-column"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="join-table" type="orm:join-table"
      minOccurs="0"/>
  </xsd:choice>
  <xsd:element name="cascade" type="orm:cascade-type"
    minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="name" type="xsd:string" use="required"/>
<xsd:attribute name="target-entity" type="xsd:string"/>
<xsd:attribute name="fetch" type="orm:fetch-type"/>
<xsd:attribute name="optional" type="xsd:boolean"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="cascade-type">
  <xsd:annotation>
    <xsd:documentation>
      public enum CascadeType { ALL, PERSIST, MERGE, REMOVE, REFRESH};
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="cascade-all" type="orm:emptyType"
      minOccurs="0"/>
    <xsd:element name="cascade-persist" type="orm:emptyType"
      minOccurs="0"/>
    <xsd:element name="cascade-merge" type="orm:emptyType"
      minOccurs="0"/>
    <xsd:element name="cascade-remove" type="orm:emptyType"
      minOccurs="0"/>
    <xsd:element name="cascade-refresh" type="orm:emptyType"
      minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="one-to-one">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface OneToOne {
        Class targetEntity() default void.class;
        CascadeType[] cascade() default {};
        FetchType fetch() default EAGER;
        boolean optional() default true;
        String mappedBy() default "";
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="primary-key-join-column"
        type="orm:primary-key-join-column"
        minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="join-column" type="orm:join-column"
        minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="join-table" type="orm:join-table"
        minOccurs="0"/>
    </xsd:choice>
    <xsd:element name="cascade" type="orm:cascade-type"
      minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="target-entity" type="xsd:string"/>
  <xsd:attribute name="fetch" type="orm:fetch-type"/>
  <xsd:attribute name="optional" type="xsd:boolean"/>
  <xsd:attribute name="mapped-by" type="xsd:string"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="one-to-many">
  <xsd:annotation>
    <xsd:documentation>

```

```

        @Target({METHOD, FIELD}) @Retention(RUNTIME)
        public @interface OneToMany {
            Class targetEntity() default void.class;
            CascadeType[] cascade() default {};
            FetchType fetch() default LAZY;
            String mappedBy() default "";
        }
    }
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
    <xsd:element name="order-by" type="orm:order-by"
        minOccurs="0"/>
    <xsd:element name="map-key" type="orm:map-key"
        minOccurs="0"/>
    <xsd:choice>
        <xsd:element name="join-table" type="orm:join-table"
            minOccurs="0"/>
        <xsd:element name="join-column" type="orm:join-column"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:choice>
    <xsd:element name="cascade" type="orm:cascade-type"
        minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="name" type="xsd:string" use="required"/>
<xsd:attribute name="target-entity" type="xsd:string"/>
<xsd:attribute name="fetch" type="orm:fetch-type"/>
<xsd:attribute name="mapped-by" type="xsd:string"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="join-table">
    <xsd:annotation>
        <xsd:documentation>

            @Target({METHOD, FIELD}) @Retention(RUNTIME)
            public @interface JoinTable {
                String name() default "";
                String catalog() default "";
                String schema() default "";
                JoinColumn[] joinColumns() default {};
                JoinColumn[] inverseJoinColumns() default {};
                UniqueConstraint[] uniqueConstraints() default {};
            }

        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="join-column" type="orm:join-column"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="inverse-join-column" type="orm:join-column"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="unique-constraint" type="orm:unique-constraint"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="catalog" type="xsd:string"/>
    <xsd:attribute name="schema" type="xsd:string"/>
</xsd:complexType>

<!-- ***** -->

<xsd:complexType name="many-to-many">
    <xsd:annotation>
        <xsd:documentation>

            @Target({METHOD, FIELD}) @Retention(RUNTIME)
            public @interface ManyToMany {
                Class targetEntity() default void.class;
                CascadeType[] cascade() default {};
                FetchType fetch() default LAZY;
                String mappedBy() default "";
            }

        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="order-by" type="orm:order-by"
            minOccurs="0"/>
        <xsd:element name="map-key" type="orm:map-key"
            minOccurs="0"/>
        <xsd:element name="join-table" type="orm:join-table"
            minOccurs="0"/>
        <xsd:element name="cascade" type="orm:cascade-type"
            minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="target-entity" type="xsd:string"/>
    <xsd:attribute name="fetch" type="orm:fetch-type"/>
    <xsd:attribute name="mapped-by" type="xsd:string"/>
</xsd:complexType>

```

```

<!-- ***** -->
<xsd:complexType name="generated-value">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface GeneratedValue {
        GenerationType strategy() default AUTO;
        String generator() default "";
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="strategy" type="orm:generation-type"/>
  <xsd:attribute name="generator" type="xsd:string"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="map-key">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface MapKey {
        String name() default "";
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>

<!-- ***** -->
<xsd:simpleType name="order-by">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface OrderBy {
        String value() default "";
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<!-- ***** -->
<xsd:complexType name="inheritance">
  <xsd:annotation>
    <xsd:documentation>
      @Target({TYPE}) @Retention(RUNTIME)
      public @interface Inheritance {
        InheritanceType strategy() default SINGLE_TABLE;
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="strategy" type="orm:inheritance-type"/>
</xsd:complexType>

<!-- ***** -->
<xsd:simpleType name="inheritance-type">
  <xsd:annotation>
    <xsd:documentation>
      public enum InheritanceType
      { SINGLE_TABLE, JOINED, TABLE_PER_CLASS};
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="SINGLE_TABLE"/>
    <xsd:enumeration value="JOINED"/>
    <xsd:enumeration value="TABLE_PER_CLASS"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- ***** -->
<xsd:simpleType name="discriminator-value">
  <xsd:annotation>
    <xsd:documentation>
      @Target({TYPE}) @Retention(RUNTIME)
      public @interface DiscriminatorValue {

```

```

        String value();
    }
</xsd:documentation>
</xsd:annotation>
<xsd:restriction base="xsd:string"/>
</xsd:simpleType>
<!-- ***** -->
<xsd:simpleType name="discriminator-type">
<xsd:annotation>
<xsd:documentation>
    public enum DiscriminatorType { STRING, CHAR, INTEGER };
</xsd:documentation>
</xsd:annotation>
<xsd:restriction base="xsd:token">
<xsd:enumeration value="STRING"/>
<xsd:enumeration value="CHAR"/>
<xsd:enumeration value="INTEGER"/>
</xsd:restriction>
</xsd:simpleType>
<!-- ***** -->
<xsd:complexType name="primary-key-join-column">
<xsd:annotation>
<xsd:documentation>
    @Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
    public @interface PrimaryKeyJoinColumn {
        String name() default "";
        String referencedColumnName() default "";
        String columnDefinition() default "";
    }
</xsd:documentation>
</xsd:annotation>
<xsd:attribute name="name" type="xsd:string"/>
<xsd:attribute name="referenced-column-name" type="xsd:string"/>
<xsd:attribute name="column-definition" type="xsd:string"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="discriminator-column">
<xsd:annotation>
<xsd:documentation>
    @Target({TYPE}) @Retention(RUNTIME)
    public @interface DiscriminatorColumn {
        String name() default "DTYPE";
        DiscriminatorType discriminatorType() default STRING;
        String columnDefinition() default "";
        int length() default 31;
    }
</xsd:documentation>
</xsd:annotation>
<xsd:attribute name="name" type="xsd:string"/>
<xsd:attribute name="discriminator-type" type="orm:discriminator-type"/>
<xsd:attribute name="column-definition" type="xsd:string"/>
<xsd:attribute name="length" type="xsd:int"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="embeddable">
<xsd:annotation>
<xsd:documentation>
    Defines the settings and mappings for embeddable objects. Is
    allowed to be sparsely populated and used in conjunction with
    the annotations. Alternatively, the metadata-complete attribute
    can be used to indicate that no annotations are to be processed
    in the class. If this is the case then the defaulting rules will
    be recursively applied.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
    @Target({TYPE}) @Retention(RUNTIME)
    public @interface Embeddable {}
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description" type="xsd:string" minOccurs="0"/>
<xsd:element name="attributes" type="orm:embeddable-attributes"
    minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="class" type="xsd:string" use="required"/>
<xsd:attribute name="access" type="orm:access-type"/>
<xsd:attribute name="metadata-complete" type="xsd:boolean"/>

```

```

</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="embeddable-attributes">
  <xsd:sequence>
    <xsd:element name="basic" type="orm:basic"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="transient" type="orm:transient"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="embedded">
  <xsd:annotation>
    <xsd:documentation>
      @Target({METHOD, FIELD}) @Retention(RUNTIME)
      public @interface Embedded {}
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="attribute-override" type="orm:attribute-override"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="mapped-superclass">
  <xsd:annotation>
    <xsd:documentation>
      Defines the settings and mappings for a mapped superclass. Is
      allowed to be sparsely populated and used in conjunction with
      the annotations. Alternatively, the metadata-complete attribute
      can be used to indicate that no annotations are to be processed
      If this is the case then the defaulting rules will be recursively
      applied.
      @Target(TYPE) @Retention(RUNTIME)
      public @interface MappedSuperclass{}
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    <xsd:element name="id-class" type="orm:id-class" minOccurs="0"/>
    <xsd:element name="exclude-default-listeners" type="orm:emptyType"
      minOccurs="0"/>
    <xsd:element name="exclude-superclass-listeners" type="orm:emptyType"
      minOccurs="0"/>
    <xsd:element name="entity-listeners" type="orm:entity-listeners"
      minOccurs="0"/>
    <xsd:element name="pre-persist" type="orm:pre-persist" minOccurs="0"/>
    <xsd:element name="post-persist" type="orm:post-persist"
      minOccurs="0"/>
    <xsd:element name="pre-remove" type="orm:pre-remove" minOccurs="0"/>
    <xsd:element name="post-remove" type="orm:post-remove" minOccurs="0"/>
    <xsd:element name="pre-update" type="orm:pre-update" minOccurs="0"/>
    <xsd:element name="post-update" type="orm:post-update" minOccurs="0"/>
    <xsd:element name="post-load" type="orm:post-load" minOccurs="0"/>
    <xsd:element name="attributes" type="orm:attributes" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="class" type="xsd:string" use="required"/>
  <xsd:attribute name="access" type="orm:access-type"/>
  <xsd:attribute name="metadata-complete" type="xsd:boolean"/>
</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="sequence-generator">
  <xsd:annotation>
    <xsd:documentation>
      @Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
      public @interface SequenceGenerator {
        String name();
        String sequenceName() default "";
        int initialValue() default 1;
        int allocationSize() default 50;
      }
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="sequence-name" type="xsd:string"/>
  <xsd:attribute name="initial-value" type="xsd:int"/>
  <xsd:attribute name="allocation-size" type="xsd:int"/>

```

```

</xsd:complexType>
<!-- ***** -->
<xsd:complexType name="table-generator">
  <xsd:annotation>
    <xsd:documentation>

      @Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
      public @interface TableGenerator {
        String name();
        String table() default "";
        String catalog() default "";
        String schema() default "";
        String pkColumnName() default "";
        String valueColumnName() default "";
        String pkColumnValue() default "";
        int initialValue() default 0;
        int allocationSize() default 50;
        UniqueConstraint[] uniqueConstraints() default {};
      }

    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="unique-constraint" type="orm:unique-constraint"
      minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required" />
  <xsd:attribute name="table" type="xsd:string" />
  <xsd:attribute name="catalog" type="xsd:string" />
  <xsd:attribute name="schema" type="xsd:string" />
  <xsd:attribute name="pk-column-name" type="xsd:string" />
  <xsd:attribute name="value-column-name" type="xsd:string" />
  <xsd:attribute name="pk-column-value" type="xsd:string" />
  <xsd:attribute name="initial-value" type="xsd:int" />
  <xsd:attribute name="allocation-size" type="xsd:int" />
</xsd:complexType>
</xsd:schema>

```

5.4. Conclusion

That exhausts persistence metadata annotations. We present the class definitions for our sample model below:

Example 5.2. Complete Metadata

```

package org.mag;

@Entity
@IdClass(Magazine.MagazineId.class)
public class Magazine
{
    @Id private String isbn;
    @Id private String title;
    @Version private int version;

    private double price; // defaults to @Basic
    private int copiesSold; // defaults to @Basic

    @OneToOne(fetch=FetchType.LAZY,
      cascade={CascadeType.PERSIST,CascadeType.REMOVE})
    private Article coverArticle;

    @OneToMany(cascade={CascadeType.PERSIST,CascadeType.REMOVE})
    @OrderBy
    private Collection<Article> articles;

    @ManyToOne(fetch=FetchType.LAZY, cascade=CascadeType.PERSIST)
    private Company publisher;

    @Transient private byte[] data;

    ...

    public static class MagazineId
    {
        ...
    }
}

```

```
}

@Entity
public class Article
{
    @Id private long id;
    @Version private int version;

    private String title; // defaults to @Basic
    private byte[] content; // defaults to @Basic

    @ManyToMany(cascade=CascadeType.PERSIST)
    @OrderBy("lastName, firstName")
    private Collection<Author> authors;

    ...
}

package org.mag.pub;

@Entity
public class Company
{
    @Id private long id;
    @Version private int version;

    private String name; // defaults to @Basic
    private double revenue; // defaults to @Basic
    private Address address; // defaults to @Embedded

    @OneToMany(mappedBy="publisher", cascade=CascadeType.PERSIST)
    private Collection<Magazine> mags;

    @OneToMany(cascade={CascadeType.PERSIST,CascadeType.REMOVE})
    private Collection<Subscription> subscriptions;

    ...
}

@Entity
public class Author
{
    @Id private long id;
    @Version private int version;

    private String firstName; // defaults to @Basic
    private double lastName; // defaults to @Basic
    private Address address; // defaults to @Embedded

    @ManyToMany(mappedBy="authors", cascade=CascadeType.PERSIST)
    private Collection<Article> arts;

    ...
}

@Embeddable
public class Address
{
    private String street; // defaults to @Basic
    private String city; // defaults to @Basic
    private String state; // defaults to @Basic
    private String zip; // defaults to @Basic

    ...
}

package org.mag.subscribe;

@MappedSuperclass
public abstract class Document
{
    @Id private long id;
    @Version private int version;

    ...
}

@Entity
public class Contract
    extends Document
{
    private String terms; // defaults to @Basic

    ...
}

@Entity
public class Subscription
{
    @Id private long id;
    @Version private int version;
```

```

private Date startDate; // defaults to @Basic
private double payment; // defaults to @Basic

@OneToMany(cascade={CascadeType.PERSIST,CascadeType.REMOVE})
@MapKey(name="num")
private Map<Long,LineItem> lineItems;

...

@Entity
public static class LineItem
    extends Contract
{
    private String comments; // defaults to @Basic
    private double price;    // defaults to @Basic
    private long num;        // defaults to @Basic

    @ManyToOne
    private Magazine magazine;

    ...
}

@Entity(name="Lifetime")
public class LifetimeSubscription
    extends Subscription
{
    @Basic(fetch=FetchType.LAZY)
    private boolean getEliteClub () { ... }
    public void setEliteClub (boolean elite) { ... }

    ...
}

@Entity(name="Trial")
public class TrialSubscription
    extends Subscription
{
    public Date getEndDate () { ... }
    public void setEndDate (Date end) { ... }

    ...
}

```

The same metadata declarations in XML:

```

<entity-mappings>
  <!-- declares a default access type for all entities -->
  <access-type>FIELD</access-type>
  <mapped-superclass class="org.mag.subscribe.Document">
    <attributes>
      <id name="id">
        <generated-value strategy="IDENTITY"/>
      </id>
      <version name="version"/>
    </attributes>
  </mapped-superclass>
  <entity class="org.mag.Magazine">
    <id-class="org.mag.Magazine$MagazineId"/>
    <attributes>
      <id name="isbn"/>
      <id name="title"/>
      <basic name="name"/>
      <basic name="price"/>
      <basic name="copiesSold"/>
      <version name="version"/>
      <many-to-one name="publisher" fetch="LAZY">
        <cascade>
          <cascade-persist/>
        </cascade>
      </many-to-one>
      <one-to-many name="articles">
        <order-by/>
        <cascade>
          <cascade-persist/>
          <cascade-remove/>
        </cascade>
      </one-to-many>
      <one-to-one name="coverArticle" fetch="LAZY">
        <cascade>
          <cascade-persist/>
          <cascade-remove/>
        </cascade>
      </one-to-one>
    </attributes>
  </entity>

```

```

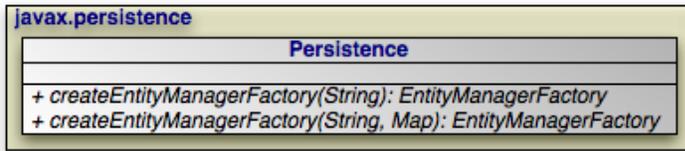
        <transient name="data" />
    </attributes>
</entity>
<entity class="org.mag.Article">
    <attributes>
        <id name="id" />
        <basic name="title" />
        <basic name="content" />
        <version name="version" />
        <many-to-many name="articles">
            <order-by>lastName, firstName</order-by>
        </many-to-many>
    </attributes>
</entity>
<entity class="org.mag.pub.Company">
    <attributes>
        <id name="id" />
        <basic name="name" />
        <basic name="revenue" />
        <version name="version" />
        <one-to-many name="mags" mapped-by="publisher">
            <cascade>
                <cascade-persist />
            </cascade>
        </one-to-many>
        <one-to-many name="subscriptions">
            <cascade>
                <cascade-persist />
                <cascade-remove />
            </cascade>
        </one-to-many>
    </attributes>
</entity>
<entity class="org.mag.pub.Author">
    <attributes>
        <id name="id" />
        <basic name="firstName" />
        <basic name="lastName" />
        <version name="version" />
        <many-to-many name="arts" mapped-by="authors">
            <cascade>
                <cascade-persist />
            </cascade>
        </many-to-many>
    </attributes>
</entity>
<entity class="org.mag.subscribe.Contract">
    <attributes>
        <basic name="terms" />
    </attributes>
</entity>
<entity class="org.mag.subscribe.Subscription">
    <attributes>
        <id name="id" />
        <basic name="payment" />
        <basic name="startDate" />
        <version name="version" />
        <one-to-many name="items">
            <map-key name="num">
                <cascade>
                    <cascade-persist />
                    <cascade-remove />
                </cascade>
            </one-to-many>
        </attributes>
</entity>
<entity class="org.mag.subscribe.Subscription.LineItem">
    <attributes>
        <basic name="comments" />
        <basic name="price" />
        <basic name="num" />
        <many-to-one name="magazine" />
    </attributes>
</entity>
<entity class="org.mag.subscribe.LifetimeSubscription" name="Lifetime">
    access="PROPERTY">
    <attributes>
        <basic name="eliteClub" fetch="LAZY" />
    </attributes>
</entity>
<entity class="org.mag.subscribe.TrialSubscription" name="Trial">
    <attributes>
        <basic name="endDate" />
    </attributes>
</entity>
<embeddable class="org.mag.pub.Address">
    <attributes>
        <basic name="street" />
        <basic name="city" />
        <basic name="state" />
        <basic name="zip" />
    </attributes>
</embeddable>

```

```
</entity-mappings>
```

Chapter 12, *Mapping Metadata* [118] will show you how to map your persistent classes to the datastore using additional annotations and XML markup. First, however, we turn to the JPA runtime APIs.

Chapter 6. Persistence



Note

Kodo also includes the `KodoPersistence` helper class to provide additional utility methods.

Within a container, you will typically use *injection* to access an `EntityManagerFactory`. Applications operating in a container, however, can use the `Persistence` class to obtain `EntityManagerFactory` objects in a vendor-neutral fashion.

```
public static EntityManagerFactory createEntityManagerFactory (String name);
public static EntityManagerFactory createEntityManagerFactory (String name, Map props);
```

Each `createEntityManagerFactory` method searches the system for an `EntityManagerFactory` definition with the given name. Use `null` for an unnamed factory. The optional map contains vendor-specific property settings used to further configure the factory.

`persistence.xml` files define `EntityManagerFactories`. The `createEntityManagerFactory` methods search for `persistence.xml` files within the `META-INF` directory of any `CLASSPATH` element. For example, if your `CLASSPATH` contains the `conf` directory, you could place an `EntityManagerFactory` definition in `conf/META-INF/persistence.xml`.

6.1. persistence.xml

The `persistence.xml` file format obeys the following Document Type Descriptor (DTD):

```
<!ELEMENT persistence (persistence-unit*)>
<!ELEMENT persistence-unit (description?,provider?,jta-datasource?,
    non-jta-datasource?,(class|jar-file|mapping-file)*,
    exclude-unlisted-classes?,properties?)>
<!ATTLIST persistence-unit name CDATA #REQUIRED>
<!ATTLIST persistence-unit transaction-type (JTA|RESOURCE_LOCAL) "JTA">
<!ELEMENT description (#PCDATA)>
<!ELEMENT provider (#PCDATA)>
<!ELEMENT jta-datasource (#PCDATA)>
<!ELEMENT non-jta-datasource (#PCDATA)>
<!ELEMENT mapping-file (#PCDATA)>
<!ELEMENT jar-file (#PCDATA)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT exclude-unlisted-classes EMPTY>
<!ELEMENT properties (property*)>
<!ELEMENT property EMPTY>
<!ATTLIST property name CDATA #REQUIRED>
<!ATTLIST property value CDATA #REQUIRED>
```

The root element of a `persistence.xml` file is `persistence`, which then contains one or more `persistence-unit` definitions. Each persistence unit describes the configuration for the entity managers created by the persistence unit's entity manager factory. The persistence unit can specify these elements and attributes.

- `name`: This is the name you pass to the `Persistence.createEntityManagerFactory` methods described above. The name attribute is required.
- `transaction-type`: Whether to use managed (JTA) or local (`RESOURCE_LOCAL`) transaction management. The default is JTA.
- `provider`: If you are using a third-party JPA vendor, this element names its implementation of the `PersistenceProvider` bootstrapping interface.

Note

Set the provider to `kodo.persistence.PersistenceProviderImpl` to use Kodo.

- `jta-data-source`: The JNDI name of a JDBC `DataSource` that is automatically enlisted in JTA transactions. This may be an XA `DataSource`.
- `non-jta-data-source`: The JNDI name of a JDBC `DataSource` that is not enlisted in JTA transactions.
- `mapping-file*`: The resource names of XML mapping files for entities and embeddable classes. You can also specify mapping information in an `orm.xml` file in your `META-INF` directory. If present, the `orm.xml` mapping file will be read automatically.
- `jar-file*`: The names of jar files containing entities and embeddable classes. The implementation will scan the jar for annotated classes.
- `class*`: The class names of entities and embeddable classes.
- `properties`: This element contains nested `property` elements used to specify vendor-specific settings. Each `property` has a name attribute and a value attribute.

Note

The Reference Guide's [Chapter 2, Configuration \[393\]](#) describes Kodo's configuration properties.

Here is a typical `persistence.xml` file for a non-EE environment:

Example 6.1. `persistence.xml`

```
<?xml version="1.0"?>
<persistence>
  <persistence-unit name="kodo">
    <provider>kodo.persistence.PersistenceProviderImpl</provider>
    <class>tutorial.Animal</class>
    <class>tutorial.Dog</class>
    <class>tutorial.Rabbit</class>
    <class>tutorial.Snake</class>
    <properties>
      <property name="kodo.ConnectionURL" value="jdbc:hsqldb:tutorial_database"/>
      <property name="kodo.ConnectionDriverName" value="org.hsqldb.jdbcDriver"/>
      <property name="kodo.ConnectionUserName" value="sa"/>
      <property name="kodo.ConnectionPassword" value=""/>
      <property name="kodo.Log" value="DefaultLevel=WARN, Tool=INFO"/>
    </properties>
  </persistence-unit>
</persistence>
```

6.2. Non-EE Use

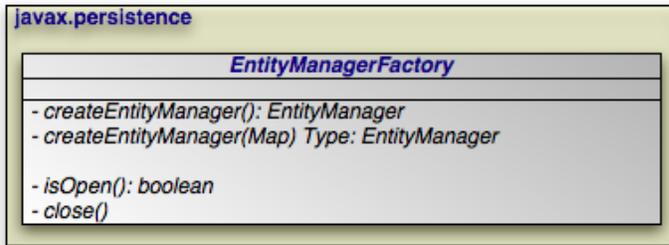
The example below demonstrates the `Persistence` class in action. You will typically execute code like this on application startup, then cache the resulting factory for future use. This bootstrapping code is only necessary in non-EE environments; in an EE environment `EntityManagerFactories` are typically injected.

Example 6.2. Obtaining an EntityManagerFactory

```
// if your persistence.xml file does not contain all settings already, you
// can add vendor settings to a map
Properties props = new Properties ();
...

// create the factory defined by the "kodo" entity-manager entry
EntityManagerFactory emf = Persistence.createEntityManagerFactory ("kodo", props);
```

Chapter 7. EntityManagerFactory



The EntityManagerFactory creates EntityManager instances for application use.

Note

Kodo extends the standard EntityManagerFactory interface with the `KodoEntityManagerFactory` to provide additional functionality.

7.1. Obtaining an EntityManagerFactory

Within a container, you will typically use *injection* to access an EntityManagerFactory. There are, however, alternative mechanisms for EntityManagerFactory construction.

Some vendors may supply public constructors for their EntityManagerFactory implementations, but we recommend: using the Java Connector Architecture (JCA) in a managed environment; using the Persistence class's createEntityManagerFactory methods in an unmanaged environment, as described in [Chapter 6, Persistence \[63\]](#). These strategies allow vendors to pool factories, cutting down on resource utilization.

JPA allows you to create and configure an EntityManagerFactory, then store it in a Java Naming and Directory Interface (JNDI) tree for later retrieval and use.

7.2. Obtaining EntityManagers

```
public EntityManager createEntityManager ();
public EntityManager createEntityManager (Map map);
```

The two createEntityManager methods above create a new EntityManager each time they are invoked. The optional Map is used to supply vendor-specific settings. If you have configured your implementation for JTA transactions and a JTA transaction is active, the returned EntityManager will be synchronized with that transaction.

Note

Kodo recognizes the following string keys in the map supplied to createEntityManager:

- kodo.ConnectionUserName
- kodo.ConnectionPassword

- `kodo.ConnectionRetainMode`
- `kodo.TransactionMode`
- `kodo.<property>`, where *<property>* is any JavaBean property of the `kodo.persistence.KodoEntityManager`.

The last option uses reflection to configure any property of Kodo's `EntityManager` implementation with the value supplied in your map. The first options correspond exactly to the same-named Kodo configuration keys described in [Chapter 2, Configuration \[393\]](#) of the Reference Guide.

7.3. Persistence Context

A persistence context is a set of entities such that for any persistent identity there is a unique entity instance. Within a persistence context, entities are *managed*. The `EntityManager` controls their lifecycle, and they can access datastore resources.

When a persistence context ends, previously-managed entities become *detached*. A detached entity is no longer under the control of the `EntityManager`, and no longer has access to datastore resources. We discuss detachment in detail in [Section 8.2, “Entity Lifecycle Management” \[71\]](#). For now, it is sufficient to know that detachment has two obvious consequences:

1. The detached entity cannot load any additional persistent state.
2. The `EntityManager` will not return the detached entity from `find`, nor will queries include the detached entity in their results. Instead, `find` method invocations and query executions that would normally incorporate the detached entity will create a new managed entity with the same identity.

Note

Kodo offers several features related to detaching entities. See [Section 11.1, “Detach and Attach” \[577\]](#) of the Reference Guide. In particular, [Section 11.1.3, “Defining the Detached Object Graph” \[578\]](#) describes how to use the `Detach-State` setting to boost the performance of merging detached entities.

Injected `EntityManager`s use a *transaction* persistence context, while `EntityManager`s obtained through the `EntityManagerFactory` have an *extended* persistence context. We describe these persistence context types below.

7.3.1. Transaction Persistence Context

Under the transaction persistence context model, an `EntityManager` begins a new persistence context with each transaction, and ends the context when the transaction commits or rolls back. Within the transaction, entities you retrieve through the `EntityManager` or via `Queries` are managed entities. They can access datastore resources to lazy-load additional persistent state as needed, and only one entity may exist for any persistent identity.

When the transaction completes, all entities lose their association with the `EntityManager` and become detached. Traversing a persistent field that wasn't already loaded now has undefined results. And using the `EntityManager` or a `Query` to retrieve additional objects may now create new instances with the same persistent identities as detached instances.

If you use an `EntityManager` with a transaction persistence context model outside of an active transaction, each method invocation creates a new persistence context, performs the method action, and ends the persistence context. For example, consider using the `EntityManager.find` method outside of a transaction. The `EntityManager` will create a temporary persistence context, perform the `find` operation, end the persistence context, and return the detached result object to you. A second call with the same id will return a second detached object.

When the next transaction begins, the `EntityManager` will begin a new persistence context, and will again start returning managed entities. As you'll see in **Chapter 8, *EntityManager* [70]** you can also merge the previously-detached entites back into the new persistence context.

Example 7.1. Behavior of Transaction Persistence Context

The following code illustrates the behavior of entites under an `EntityManager` using a transaction persistence context.

```
EntityManager em; // Injected
...

// Outside a transaction:

// Each operation occurs in a separate persistence context and returns
// a new detached instance.
Magazine mag1 = em.find (Magazine.class, magId);
Magazine mag2 = em.find (Magazine.class, magId);
assertTrue (mag2 != mag1);
...

// Transaction begins:

// Within a transaction, a subsequent lookup doesn't return any of the
// detached objects. However, two lookups within the same transaction
// return the same instanc, because the persistence context spans the
// transaction.
Magazine mag3 = em.find (Magazine.class, magId);
assertTrue (mag3 != mag1 && mag3 != mag2);
Magazine mag4 = em.find (Magazine.class (magId);
assertTrue (mag4 == mag3);
...

// Transaction commits:

// Once again, each operation returns a new instance.
Magazine mag5 = em.find (Magazine.class, magId);
assertTrue (mag5 != mag3);
```

7.3.2. Extended Persistence Context

An `EntityManager` using an extended persistence context maintains the same persistence context for its entire lifecycle. Whether inside a transaction or not, all entities returned from the `EntityManager` are managed, and the `EntityManager` never creates two entity instances to represent the same persistent identity. Entities only become detached when you finally close the `EntityManager` (or when they are serialized).

Example 7.2. Behavior of Extended Persistence Context

The following code illustrates the behavior of entites under an `EntityManager` using an extended persistence context.

```
EntityManagerFactory emf = ...
EntityManager em = emf.createEntityManager (PersistenceContextType.EXTENDED);

// The persistence context is active for the entire life of the EM, so there is
// only one entity for a given persistent identity.
Magazine mag1 = em.find (Magazine.class, magId);
Magazine mag2 = em.find (Magazine.class, magId);
assertTrue (mag2 == mag1);

em.getTransaction ().begin ();

// The same persistence context is active within the transaction.
Magazine mag3 = em.find (Magazine.class, magId);
assertTrue (mag3 == mag1);
Magazine mag4 = em.find (Magazine.class (magId);
assertTrue (mag4 == mag1);

em.getTransaction.commit ();
```

```
// When the transaction commits, the instance is still managed.
Magazine mag5 = em.find (Magazine.class, magId);
assertTrue (mag5 == mag1);

// The instance finally becomes detached when the EM closes.
em.close ();
```

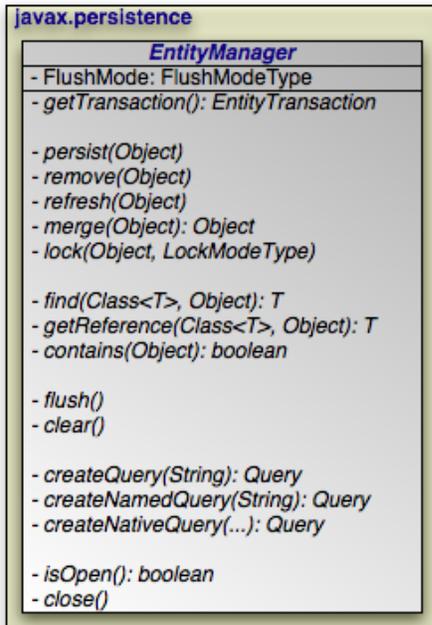
7.4. Closing the EntityManagerFactory

```
public boolean isOpen ();
public void close ();
```

`EntityManagerFactory` instances are heavyweight objects. Each factory might maintain a metadata cache, object state cache, `EntityManager` pool, connection pool, and more. If your application no longer needs an `EntityManagerFactory`, you should close it to free these resources. When an `EntityManagerFactory` closes, all `EntityManagers` from that factory, and by extension all entities managed by those `EntityManagers`, become invalid. Attempting to close an `EntityManagerFactory` while one or more of its `EntityManagers` has an active transaction may result in an `IllegalStateException`.

Closing an `EntityManagerFactory` should not be taken lightly. It is much better to keep a factory open for a long period of time than to repeatedly create and close new factories. Thus, most applications will never close the factory, or only close it when the application is exiting. Only applications that require multiple factories with different configurations have an obvious reason to create and close multiple `EntityManagerFactory` instances. Once a factory is closed, all methods except `isOpen` throw an `IllegalStateException`.

Chapter 8. EntityManager



The diagram above presents an overview of the `EntityManager` interface. For a complete treatment of the `EntityManager` API, see the **Javadoc** documentation. Methods whose parameter signatures consist of an ellipsis (...) are overloaded to take multiple parameter types.

Note

Kodo extends the standard `EntityManager` interface with the `kodo.persistence.KodoEntityManager` interface to provide additional functionality.

The `EntityManager` is the primary interface used by application developers to interact with the EJB persistence runtime. The methods of the `EntityManager` can be divided into the following functional categories:

- Transaction association
- Entity lifecycle management
- Entity identity management
- Cache management
- Query factory
- Closing

8.1. Transaction Association

```
public EntityTransaction getTransaction ();
```

Every `EntityManager` has a one-to-one relation with an `EntityTransaction` instance. In fact, many vendors use a single class to implement both the `EntityManager` and `EntityTransaction` interfaces. If your application requires multiple concurrent transactions, you will use multiple `EntityManager`s.

You can retrieve the `EntityTransaction` associated with an `EntityManager` through the `getTransaction` method. Note that most EJB persistence implementations can integrate with an application server's managed transactions. If you take advantage of this feature, you will control transactions by declarative demarcation or through the Java Transaction API (JTA) rather than through the `EntityTransaction`.

8.2. Entity Lifecycle Management

`EntityManager`s perform several actions that affect the lifecycle state of entity instances.

```
public void persist (Object entity);
```

The above method transitions new instances to managed instances. On the next flush or commit, the newly persisted instances will be inserted into the datastore.

For a given entity A, the `persist` method behaves as follows:

- If A is a new entity, it becomes managed.
- If A is an existing managed entity, it is ignored. However, the `persist` operation cascades as defined below.
- If A is a removed entity, it becomes managed.
- If A is a detached entity, an `IllegalArgumentException` is thrown.
- The `persist` operation recurses on all relation fields of A whose **cascades** include `CascadeType.PERSIST`.

This action can only be used in the context of an active transaction.

```
public void remove (Object entity);
```

The above method transitions managed instances to removed instances. The instances will be deleted from the datastore on the next flush or commit. Accessing a removed entity has undefined results.

For a given entity A, the `remove` method behaves as follows:

- If A is a new entity, it is ignored. However, the `remove` operation cascades as defined below.
- If A is an existing managed entity, it becomes removed.
- If A is a removed entity, it is ignored.
- If A is a detached entity, an `IllegalArgumentException` is thrown.
- The `remove` operation recurses on all relation fields of A whose **cascades** include `CascadeType.REMOVE`.

This action can only be used in the context of an active transaction.

```
public void refresh (Object entity);
```

Use the `refresh` method to make sure the persistent state of an instance is synchronized with the values in the datastore. `refresh` is intended for long-running optimistic transactions in which there is a danger of seeing stale data.

For a given entity A, the `refresh` method behaves as follows:

- If A is a new entity, it is ignored. However, the remove operation cascades as defined below.
- If A is an existing managed entity, its state is refreshed from the datastore.
- If A is a removed entity, it is ignored.
- If A is a detached entity, an `IllegalArgumentException` is thrown.
- The refresh operation recurses on all relation fields of A whose **cascades** include `CascadeType.REFRESH`.

```
public Object merge (Object entity);
```

A common use case for an application running in a servlet or application server is to "detach" objects from all server resources, modify them, and then "attach" them again. For example, a servlet might store persistent data in a user session between a modification based on a series of web forms. Between each form request, the web container might decide to serialize the session, requiring that the stored persistent state be disassociated from any other resources. Similarly, a client/server application might transfer persistent objects to a client via serialization, allow the client to modify their state, and then have the client return the modified data in order to be saved. This is sometimes referred to as the *data transfer object* or *value object* pattern, and it allows fine-grained manipulation of data objects without incurring the overhead of multiple remote method invocations.

EJB persistence provides support for this pattern by automatically detaching entities when they are serialized or when a persistence context ends (see [Section 7.3, "Persistence Context" \[67\]](#) for an exploration of persistence contexts). The EJB persistence *merge* API re-attaches detached entities. This allows you to detach a persistent instance, modify the detached instance offline, and merge the instance back into an `EntityManager` (either the same one that detached the instance, or a new one). The changes will then be applied to the existing instance from the datastore.

A detached entity maintains its persistent identity, but cannot load additional state from the datastore. Accessing any persistent field or property that was not loaded at the time of detachment has undefined results.

Note

Be sure to not alter the version or identity fields of detached instances if you plan on merging them later.

The `merge` method returns a managed copy of the given detached entity. Changes made to the persistent state of the detached entity are applied to this managed instance. Because merging involves changing persistent state, you can only merge within a transaction.

If you attempt to merge an instance whose representation has changed in the datastore since detachment, the merge operation will throw an exception, or the transaction in which you perform the merge will fail on commit, just as if a normal optimistic conflict were detected.

Note

Kodo offers enhancements to EJB persistence detachment functionality, including additional options to control which fields are detached. See [Section 11.1, “Detach and Attach” \[577\]](#) in the Reference Guide for details.

For a given entity *A*, the `merge` method behaves as follows:

- If *A* is a detached entity, its state is copied into existing managed instance *A'* of the same entity identity, or a new managed copy of *A* is created.
- If *A* is a new entity, a new managed entity *A'* is created and the state of *A* is copied into *A'*.
- If *A* is an existing managed entity, it is ignored. However, the merge operation still cascades as defined below.
- If *A* is a removed entity, an `IllegalArgumentException` is thrown.
- The merge operation recurses on all relation fields of *A* whose **cascades** include `CascadeType.MERGE`.

```
public void lock (Object entity, LockModeType mode);
```

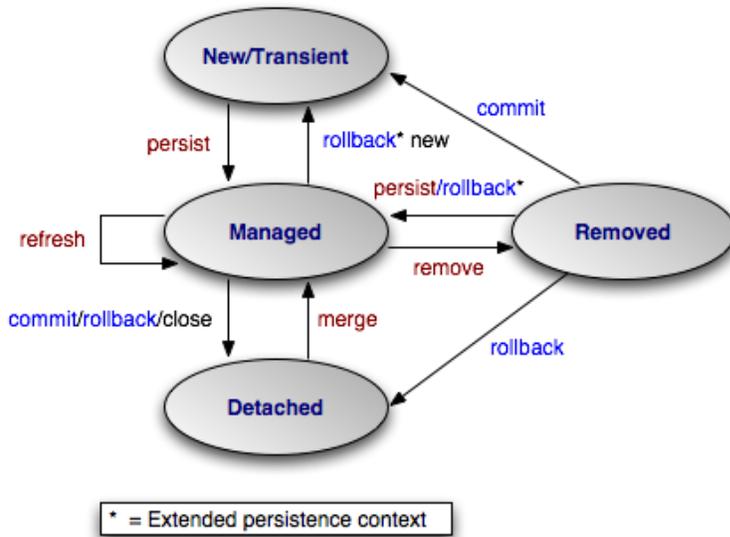
This method locks the given entity using the named mode. The `javax.persistence.LockModeType` enum defines two modes:

- **READ**: Other transactions may concurrently read the object, but cannot concurrently update it.
- **WRITE**: Other transactions cannot concurrently read or write the object. When a transaction is committed that holds **WRITE** locks on any entities, those entities will have their version incremented even if the entities themselves did not change in the transaction.

Note

Kodo has additional APIs for controlling object locking. See [Section 9.4, “Object Locking” \[542\]](#) in the Reference Guide for details.

The following diagram illustrates the lifecycle of an entity with respect to the APIs presented in this section.



8.3. Lifecycle Examples

The examples below demonstrate how to use the lifecycle methods presented in the previous section. The examples are appropriate for out-of-container use. Within a container, `EntityManager`s are usually injected, and transactions are usually managed. You would therefore omit the `createEntityManager` and `close` calls, as well as all transaction demarcation code.

Example 8.1. Persisting Objects

```

// Create some objects.
Magazine mag = new Magazine ("1B78-YU9L", "JavaWorld");

Company pub = new Company ("Weston House");
pub.setRevenue (1750000D);
mag.setPublisher (pub);
pub.addMagazine (mag);

Article art = new Article ("EJB Rules!", "Transparent Object Persistence");
art.addAuthor (new Author ("Fred", "Hoyle"));
mag.addArticle (art);

// Persist them.
EntityManager em = emf.createEntityManager ();
em.getTransaction ().begin ();
em.persist (mag);
em.persist (pub);
em.persist (art);
em.getTransaction ().commit ();

// Or we could continue using the EntityManager ...
em.close ();
    
```

Example 8.2. Updating Objects

```

Magazine.MagazineId mi = new Magazine.MagazineId ();
mi.isbn = "1B78-YU9L";
mi.title = "JavaWorld";

// Updates should always be made within transactions. Note that
    
```

```
// there is no code explicitly linking the magazine or company
// with the transaction; EJB automatically tracks all changes.
EntityManager em = emf.createEntityManager ();
em.getTransaction ().begin ();
Magazine mag = em.find (Magazine.class, mi);
mag.setPrice (5.99);
Company pub = mag.getPublisher ();
pub.setRevenue (1750000D);
em.getTransaction ().commit ();

// Or we could continue using the EntityManager ...
em.close ();
```

Example 8.3. Removing Objects

```
// Assume we have an object id for the company whose subscriptions
// we want to delete.
Object oid = ...;

// Deletes should always be made within transactions.
EntityManager em = emf.createEntityManager ();
em.getTransaction ().begin ();
Company pub = (Company) em.find (Company.class, oid);
for (Subscription sub : pub.getSubscriptions ())
    em.remove (sub);
pub.getSubscriptions ().clear ();
em.getTransaction ().commit ();

// Or we could continue using the EntityManager ...
em.close ();
```

Example 8.4. Detaching and Merging

This example demonstrates a common client/server scenario. The client requests objects and makes changes to them, while the server handles the object lookups and transactions.

```
// CLIENT:
// Requests an object with a given oid.
Record detached = (Record) getFromServer (oid);

...

// SERVER:
// Sends object to client; object detaches on EM close.
Object oid = processClientRequest ();
EntityManager em = emf.createEntityManager ();
Record record = em.find (Record.class, oid);
em.close ();
sendToClient (record);

...

// CLIENT:
// Makes some modifications and sends back to server.
detached.setSomeField ("bar");
sendToServer (detached);

...

// SERVER:
// Merges the instance and commits the changes.
Record modified = (Record) processClientRequest ();
EntityManager em = emf.createEntityManager ();
em.getTransaction ().begin ();
Record merged = (Record) em.merge (modified);
merged.setLastModified (System.currentTimeMillis ());
merged.setModifier (getClientIdentityCode ());
em.getTransaction ().commit ();
```

```
em.close ();
```

8.4. Entity Identity Management

Each `EntityManager` is responsible for managing the persistent identities of the managed objects in the persistence context. The following methods allow you to interact with the management of persistent identities. The behavior of these methods is deeply affected by the persistence context type of the `EntityManager`; see [Section 7.3, “Persistence Context” \[67\]](#) for an explanation of persistence contexts.

```
public <T> T find (Class<T> cls, Object oid);
```

This method returns the persistent instance of the given type with the given persistent identity. If the instance is already present in the current persistence context, the cached version will be returned. Otherwise, a new instance will be constructed and loaded with state from the datastore. If no entity with the given type and identity exists in the datastore, this method returns null.

```
public <T> T getReference (Class<T> cls, Object oid);
```

This method is similar to `find`, but does not necessarily go to the database when the entity is not found in cache. The implementation may construct a *hollow* entity and return it to you instead. Hollow entities do not have any state loaded. The state only gets loaded when you attempt to access a persistent field. At that time, the implementation may throw an `EntityNotFoundException` if it discovers that the entity does not exist in the datastore. The implementation may also throw an `EntityNotFoundException` from the `getReference` method itself. Unlike `find`, `getReference` does not return null.

```
public boolean contains (Object entity);
```

Returns true if the given entity is part of the current persistence context, and false otherwise. Removed entities are not considered part of the current persistence context.

8.5. Cache Management

```
public void flush ();
```

The `flush` method writes any changes that have been made in the current transaction to the datastore. If the `EntityManager` does not already have a connection to the datastore, it obtains one for the flush and retains it for the duration of the transaction. Any exceptions during flush cause the transaction to be marked for rollback. See [Chapter 9, Transaction \[79\]](#)

Flushing requires an active transaction. If there isn't a transaction in progress, the `flush` method throws a `TransactionRequiredException`.

```
public FlushModeType getFlushMode ();
```

```
public void setFlushMode (FlushModeType flushMode);
```

The `EntityManager`'s `FlushMode` property controls whether to flush transactional changes before executing queries. This allows the query results to take into account changes you have made during the current transaction. Available `javax.persistence.FlushModeType` constants are:

- `COMMIT`: Only flush when committing, or when told to do so through the `flush` method. Query results may not take into account changes made in the current transaction.
- `AUTO`: The implementation is permitted to flush before queries to ensure that the results reflect the most recent object state.

You can also set the flush mode on individual `Query` instances.

Note

Kodo only flushes before a query if the query might be affected by data changed in the current transaction. Additionally, Kodo allows fine-grained control over flushing behavior. See the Reference Guide's [Section 4.8, “Configuring the Use of JDBC Connections” \[438\]](#)

```
public void clear ();
```

Clearing the `EntityManager` effectively ends the persistence context. All entities managed by the `EntityManager` become detached.

8.6. Query Factory

```
public Query createQuery (String query);
```

`Query` objects are used to find entities matching certain criteria. The `createQuery` method creates a query using the given EJB Query Language (JPQL) string. See [Chapter 10, JPA Query \[82\]](#) for details.

```
public Query createNamedQuery (String name);
```

This method retrieves a query defined in metadata by name. The returned `Query` instance is initialized with the information declared in metadata. For more information on named queries, read [Section 10.1.9, “Named Queries” \[90\]](#)

```
public Query createNativeQuery (String sql);
public Query createNativeQuery (String sql, Class resultCls);
public Query createNativeQuery (String sql, String resultMapping);
```

Native queries are queries in the datastore's native language. For relational databases, this is the Structured Query Language (SQL). **Chapter 11, *SQL Queries* [116]** elaborates on EJB persistence's native query support.

8.7. Closing

```
public boolean isOpen ();  
public void close ();
```

When an `EntityManager` is no longer needed, you should call its `close` method. Closing an `EntityManager` releases any resources it is using. The persistence context ends, and the entities managed by the `EntityManager` become detached. Any `Query` instances the `EntityManager` created become invalid. Calling any method other than `isOpen` on a closed `EntityManager` results in an `IllegalStateException`. You cannot close an `EntityManager` that is in the middle of a transaction.

If you are in a managed environment using injected entity managers, you should not close them.

Chapter 9. Transaction

Transactions are critical to maintaining data integrity. They are used to group operations into units of work that act in an all-or-nothing fashion. Transactions have the following qualities:

- *Atomicity*. Atomicity refers to the all-or-nothing property of transactions. Either every data update in the transaction completes successfully, or they all fail, leaving the datastore in its original state. A transaction cannot be only partially successful.
- *Consistency*. Each transaction takes the datastore from one consistent state to another consistent state.
- *Isolation*. Transactions are isolated from each other. When you are reading persistent data in one transaction, you cannot "see" the changes that are being made to that data in other transactions. Similarly, the updates you make in one transaction cannot conflict with updates made in concurrent transactions. The form of conflict resolution employed depends on whether you are using pessimistic or optimistic transactions. Both types are described later in this chapter.
- *Durability*. The effects of successful transactions are durable; the updates made to persistent data last for the lifetime of the datastore.

Together, these qualities are called the ACID properties of transactions. To understand why these properties are so important to maintaining data integrity, consider the following example:

Suppose you create an application to manage bank accounts. The application includes a method to transfer funds from one user to another, and it looks something like this:

```
public void transferFunds (User from, User to, double amnt)
{
    from.decrementAccount (amnt);
    to.incrementAccount (amnt);
}
```

Now suppose that user Alice wants to transfer 100 dollars to user Bob. No problem; you simply invoke your `transferFunds` method, supplying Alice in the `from` parameter, Bob in the `to` parameter, and `100.00` as the `amnt`. The first line of the method is executed, and 100 dollars is subtracted from Alice's account. But then, something goes wrong. An unexpected exception occurs, or the hardware fails, and your method never completes.

You are left with a situation in which the 100 dollars has simply disappeared. Thanks to the first line of your method, it is no longer in Alice's account, and yet it was never transferred to Bob's account either. The datastore is in an inconsistent state.

The importance of transactions should now be clear. If the two lines of the `transferFunds` method had been placed together in a transaction, it would be impossible for only the first line to succeed. Either the funds would be transferred properly or they would not be transferred at all, and an exception would be thrown. Money could never vanish into thin air, and the data store could never get into an inconsistent state.

9.1. Transaction Types

There are two major types of transactions: pessimistic transactions and optimistic transactions. Each type has both advantages and disadvantages.

Pessimistic transactions generally lock the datastore records they act on, preventing other concurrent transactions from using the same data. This avoids conflicts between transactions, but consumes database resources. Additionally, locking records can result in *deadlock*, a situation in which two transactions are both waiting for the other to release its locks before completing. The results of a deadlock are datastore-dependent; usually one transaction is forcefully rolled back after some specified timeout interval, and an exception is thrown.

This document will often use the term *datastore* transaction in place of *pessimistic* transaction. This is to acknowledge that some datastores do not support pessimistic semantics, and that the exact meaning of a non-optimistic JPA transaction is dependent on the datastore. Most of the time, a datastore transaction is equivalent to a pessimistic transaction.

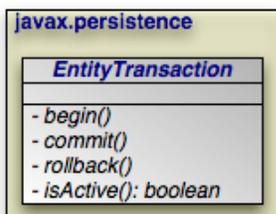
Optimistic transactions consume less resources than pessimistic/datastore transactions, but only at the expense of reliability. Because optimistic transactions do not lock datastore records, two transactions might change the same persistent information at the same time, and the conflict will not be detected until the second transaction attempts to flush or commit. At this time, the second transaction will realize that another transaction has concurrently modified the same records (usually through a timestamp or versioning system), and will throw an appropriate exception. Note that optimistic transactions still maintain data integrity; they are simply more likely to fail in heavily concurrent situations.

Despite their drawbacks, optimistic transactions are the best choice for most applications. They offer better performance, better scalability, and lower risk of hanging due to deadlock.

Note

Kodo uses optimistic semantics by default, but supports both optimistic and datastore transactions. Kodo also offers advanced locking and versioning APIs for fine-grained control over database resource allocation and object versioning. See [Section 9.4, “Object Locking” \[542\]](#) and [Section 5.8, “Lock Groups” \[476\]](#) of the Reference Guide for details on locking. [Section 5.2.5, “Version” \[35\]](#) of this document covers standard object versioning.

9.2. The EntityTransaction Interface



JPA integrates with your container's *managed* transactions, allowing you to use the container's declarative transaction demarcation and its Java Transaction API (JTA) implementation for transaction management. Outside of a container, though, you must demarcate transactions manually through JPA. The `EntityTransaction` interface controls unmanaged transactions in JPA.

Note

Kodo offers additional transaction-related functionality in `KodoEntityManager`. See [Section 8.2, “Integrating with the Transaction Manager” \[535\]](#) for how to configure Kodo to use managed transactions.

```

public void begin ();
public void commit ();
public void rollback ();
  
```

The `begin`, `commit`, and `rollback` methods demarcate transaction boundaries. The methods should be self-explanatory: `begin` starts a transaction, `commit` attempts to commit the transaction's changes to the datastore, and `rollback` aborts the transaction, in which case the datastore is "rolled back" to its previous state. JPA implementations will automatically roll back transactions if any exception is thrown during the commit process.

Unless you are using an extended persistence context, committing or rolling back also ends the persistence context. All managed entites will be detached from the `EntityManager`.

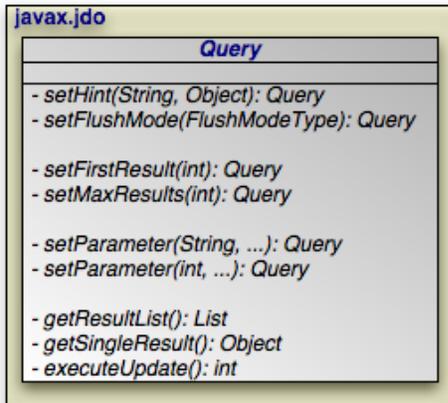
```
public boolean isActive ();
```

Finally, the `isActive` method returns `true` if the transaction is in progress (`begin` has been called more recently than `commit` or `rollback`), and `false` otherwise.

Example 9.1. Grouping Operations with Transactions

```
public void transferFunds (EntityManager em, User from, User to, double amnt)
{
    // Note: It would be better practice to move the transaction demarcation
    // code out of this method. The code below is for example purposes only.
    Transaction trans = em.getTransaction ();
    trans.begin ();
    try
    {
        from.decrementAccount (amnt);
        to.incrementAccount (amnt);
        trans.commit ();
    }
    catch (RuntimeException re)
    {
        if (trans.isActive ())
            trans.rollback (); // Alternative: Fix error and retry.
        throw re;
    }
}
```

Chapter 10. JPA Query



The `javax.persistence.Query` interface is the mechanism for issuing queries in JPA. The primary query language used is the Java Persistence Query Language, or JPQL. JPQL is syntactically very similar to SQL, but is object-oriented rather than table-oriented.

The API for executing JPQL queries will be discussed in [Section 10.1, “JPQL API” \[82\]](#) and a full language reference will be covered in [Section 10.2, “JPQL Language Reference” \[91\]](#)

10.1. JPQL API

10.1.1. Query Basics

```
SELECT x FROM Magazine x
```

The preceding is a simple JPQL query for all `Magazine` entities.

```
public Query createQuery (String jpql);
```

The `EntityManager.createQuery` method creates a `Query` instance from a given JPQL string.

```
public List getResultList ();
```

Invoking `Query.getResultList` executes the query and returns a `List` containing the matching objects. The following example executes our `Magazine` query above:

```
EntityManager em = ...
Query q = em.createQuery ("SELECT x FROM Magazine x");
List<Magazine> results = q.getResultList ();
```

A JPQL query has an internal namespace declared in the `from` clause of the query. Arbitrary identifiers are assigned to entities so that they can be referenced elsewhere in the query. In the query example above, the identifier `x` is assigned to the entity `Magazine`.

Note

The `as` keyword can optionally be used when declaring identifiers in the `from` clause. `SELECT x FROM Magazine x` and `SELECT x FROM Magazine AS x` are synonymous.

Following the `select` clause of the query is the object or objects that the query returns. In the case of the query above, the query's result list will contain instances of the `Magazine` class.

Note

When selecting entities, you can optionally use the keyword `object`. The clauses `select x` and `SELECT OBJECT(x)` are synonymous.

The optional `where` clause places criteria on matching results. For example:

```
SELECT x FROM Magazine x WHERE x.title = 'JDJ'
```

Keywords in JPQL expressions are case-insensitive, but entity, identifier, and member names are not. For example, the expression above could also be expressed as:

```
SELECT x FROM Magazine x WHERE x.title = 'JDJ'
```

But it could not be expressed as:

```
SELECT x FROM Magazine x WHERE x.TITLE = 'JDJ'
```

As with the `select` clause, alias names in the `where` clause are resolved to the entity declared in the `from` clause. The query above could be described in English as "for all `Magazine` instances `x`, return a list of every `x` such that `x`'s `title` field is equal to 'JDJ'".

JPQL uses SQL-like syntax for query criteria. The `and` and `or` logical operators chain multiple criteria together:

```
SELECT x FROM Magazine x WHERE x.title = 'JDJ' OR x.title = 'JavaPro'
```

The `=` operator tests for equality. `<>` tests for inequality. JPQL also supports the following arithmetic operators for numeric comparisons: `>`, `>=`, `<`, `<=`. For example:

```
SELECT x FROM Magazine x WHERE x.price > 3.00 AND x.price <= 5.00
```

This query returns all magazines whose price is greater than 3.00 and less than or equal to 5.00.

```
SELECT x FROM Magazine x WHERE x.price <> 3.00
```

This query returns all Magazines whose price is not equals to 3.00.

You can group expressions together using parentheses in order to specify how they are evaluated. This is similar to how parentheses are used in Java. For example:

```
SELECT x FROM Magazine x WHERE (x.price > 3.00 AND x.price <= 5.00) OR x.price = 7.00
```

This expression would match magazines whose price is 4.00, 5.00, or 7.00, but not 6.00. Alternately:

```
SELECT x FROM Magazine x WHERE x.price > 3.00 AND (x.price <= 5.00 OR x.price = 7.00)
```

This expression will magazines whose price is 5.00 or 7.00, but not 4.00 or 6.00.

JPQL also includes the following conditionals:

- [NOT] BETWEEN: Shorthand for expressing that a value falls between two other values. The following two statements are synonymous:

```
SELECT x FROM Magazine x WHERE x.price >= 3.00 AND x.price <= 5.00
```

```
SELECT x FROM Magazine x WHERE x.price BETWEEN 3.00 AND 5.00
```

- [NOT] LIKE: Performs a string comparison with wildcard support. The special character '_' in the parameter means to match any single character, and the special character '%' means to match any sequence of characters. The following statement matches title fields "JDJ" and "JavaPro", but not "IT Insider":

```
SELECT x FROM Magazine x WHERE x.title LIKE 'J%'
```

The following statement matches the title field "JDJ" but not "JavaPro":

```
SELECT x FROM Magazine x WHERE x.title LIKE 'J__'
```

- [NOT] IN: Specifies that the member must be equal to one element of the provided list. The following two statements are synonymous:

```
SELECT x FROM Magazine x WHERE x.title IN ('JDJ', 'JavaPro', 'IT Insider')
```

```
SELECT x FROM Magazine x WHERE x.title = 'JDJ' OR x.title = 'JavaPro' OR x.title = 'IT Insider'
```

- IS [NOT] EMPTY: Specifies that the collection field holds no elements. For example:

```
SELECT x FROM Magazine x WHERE x.articles is empty
```

This statement will return all magazines whose `articles` member contains no elements.

- IS [NOT] NULL: Specifies that the field is equal to null. For example:

```
SELECT x FROM Magazine x WHERE x.publisher is null
```

This statement will return all Magazine instances whose "publisher" field is set to null.

- NOT: Negates the contained expression. For example, the following two statements are synonymous:

```
SELECT x FROM Magazine x WHERE NOT(x.price = 10.0)
```

```
SELECT x FROM Magazine x WHERE x.price <> 10.0
```

10.1.2. Relation Traversal

Relations between objects can be traversed using Java-like syntax. For example, if the Magazine class has a field named "publisher" or type Company, that relation can be queried as follows:

```
SELECT x FROM Magazine x WHERE x.publisher.name = 'Random House'
```

This query returns all Magazine instances whose publisher field is set to a Company instance whose name is "Random House".

Single-valued relation traversal implies that the relation is not null. In SQL terms, this is known as an *inner join*. If you want to also include relations that are null, you can specify:

```
SELECT x FROM Magazine x WHERE x.publisher.name = 'Random House' or x.publisher is null
```

You can also traverse collection fields in queries, but you must declare each traversal in the `from` clause. Consider:

```
SELECT x FROM Magazine x, IN(x.articles) y WHERE y.authorName = 'John Doe'
```

This query says that for each `Magazine x`, traverse the `articles` relation and check each `Article y`, and pass the filter if `y`'s `authorName` field is equal to "John Doe". In short, this query will return all magazines that have any articles written by John Doe.

Note

The `IN()` syntax can also be expressed with the keywords `inner join`. The statements `SELECT x FROM Magazine x, IN(x.articles) y WHERE y.authorName = 'John Doe'` and `SELECT x FROM Magazine x inner join x.articles y WHERE y.authorName = 'John Doe'` are synonymous.

10.1.3. Fetch Joins

JPQL queries may specify one or more `join fetch` declarations, which allow the query to specify which fields in the returned instances will be pre-fetched.

```
SELECT x FROM Magazine x join fetch x.articles WHERE x.title = 'JDJ'
```

The query above returns `Magazine` instances and guarantees that the `articles` field will already be fetched in the returned instances.

Multiple fields may be specified in separate `join fetch` declarations:

```
SELECT x FROM Magazine x join fetch x.articles join fetch x.authors WHERE x.title = 'JDJ'
```

Note

Specifying the `join fetch` declaration is functionally equivalent to adding the fields to the Query's `FetchConfiguration`. See [Section 5.6, “Fetch Groups” \[469\]](#)

10.1.4. JPQL Functions

As well as supporting direct field and relation comparisons, JPQL supports a pre-defined set of functions that you can apply.

- `CONCAT(string1, string2)`: Concatenates two string fields or literals. For example:

```
SELECT x FROM Magazine x WHERE CONCAT(x.title, 's') = 'JDJs'
```

- `SUBSTRING(string, startIndex, length)`: Returns the part of the `string` argument starting at `startIndex` (1-based) and ending at `length` characters past `startIndex`.

```
SELECT x FROM Magazine x WHERE SUBSTRING(x.title, 1, 1) = 'J'
```

- `TRIM([LEADING | TRAILING | BOTH] [character FROM] string)`: Trims the specified character from either the beginning (`LEADING`), the ending (`TRAILING`), or both (`BOTH`) the beginning and ending of the string argument. If no trim character is specified, the space character will be trimmed.

```
SELECT x FROM Magazine x WHERE TRIM(BOTH 'J' FROM x.title) = 'D'
```

- `LOWER(string)`: Returns the lower-case of the specified string argument.

```
SELECT x FROM Magazine x WHERE LOWER(x.title) = 'j dj'
```

- `UPPER(string)`: Returns the upper-case of the specified string argument.

```
SELECT x FROM Magazine x WHERE UPPER(x.title) = 'JAVAPRO'
```

- `LENGTH(string)`: Returns the number of characters in the specified string argument.

```
SELECT x FROM Magazine x WHERE LENGTH(x.title) = 3
```

- `LOCATE(searchString, candidateString [, startIndex])`: Returns the first index of `searchString` in `candidateString`. Positions are 1-based. If the string is not found, returns 0.

```
SELECT x FROM Magazine x WHERE LOCATE('D', x.title) = 2
```

- `ABS(number)`: Returns the absolute value of the argument.

```
SELECT x FROM Magazine x WHERE ABS(x.price) >= 5.00
```

- `SQRT(number)`: Returns the square root of the argument.

```
SELECT x FROM Magazine x WHERE SQRT(x.price) >= 1.00
```

- `MOD(number, divisor)`: Returns the modulo of number and divisor.

```
SELECT x FROM Magazine x WHERE MOD(x.price, 10) = 0
```

- `CURRENT_DATE`: Returns the current date.
- `CURRENT_TIME`: Returns the current time.
- `CURRENT_TIMESTAMP`: Returns the current timestamp.

10.1.5. Polymorphic Queries

All JPQL queries are polymorphic, which means the `from` clause of a query includes not only instances of the specific entity class to which it refers, but all subclasses of that class as well. The instances returned by a query include instances of the subclasses that satisfy the query conditions. For example, the following query may return instances of `Magazine`, as well as `Tabloid` and `Digest` instances, where `Tabloid` and `Digest` are `Magazine` subclasses.

```
SELECT x FROM Magazine x WHERE x.price < 5
```

10.1.6. Query Parameters

JPQL provides support for parameterized queries. Either named parameters or positional parameters may be specified in the query string. Parameters allow you to re-use query templates where only the input parameters vary. A single query can declare either named parameters or positional parameters, but is not allowed to declare both named and positional parameters.

```
public Query setParameter (int pos, Object value);
```

Specify positional parameters in your JPQL string using an integer prefixed by a question mark. You can then populate the `Query` object with positional parameter values via calls to the `setParameter` method above. The method returns the `Query` instance for optional method chaining.

```
EntityManager em = ...
Query q = em.createQuery ("SELECT x FROM Magazine x WHERE x.title = ?1 and x.price > ?2");
q.setParameter (1, "JDJ").setParameter (2, 5.0);
List<Magazine> results = q.getResultList ();
```

This code will substitute JDJ for the ?1 parameter and 5.0 for the ?2 parameter, then execute the query with those values.

```
public Query setParameter (String name, Object value);
```

Named parameter are denoted by prefixing an arbitrary name with a colon in your JPQL string. You can then populate the Query object with parameter values using the method above. Like the positional parameter method, this method returns the Query instance for optional method chaining.

```
EntityManager em = ...
Query q = em.createQuery ("SELECT x FROM Magazine x WHERE x.title = :titleParam and x.price > :priceParam");
q.setParameter ("titleParam", "JDJ").setParameter ("priceParam", 5.0);
List<Magazine> results = q.getResultList ();
```

This code substitutes JDJ for the :titleParam parameter and 5.0 for the :priceParam parameter, then executes the query with those values.

10.1.7. Ordering

JPQL queries may optionally contain an `order by` clause which specifies one or more fields to order by when returning query results. You may follow the `order by field` clause with the `asc` or `desc` keywords, which indicate that ordering should be ascending or descending, respectively. If the direction is omitted, ordering is ascending by default.

```
SELECT x FROM Magazine x order by x.title asc, x.price desc
```

The query above returns `Magazine` instances sorted by their title in ascending order. In cases where the titles of two or more magazines are the same, those instances will be sorted by price in descending order.

10.1.8. Aggregates

JPQL queries can select aggregate data as well as objects. JPQL includes the `min`, `max`, `avg`, and `count` aggregates. These functions can be used for reporting and summary queries.

The following query will return the average of all the prices of all the magazines:

```
EntityManager em = ...
Query q = em.createQuery ("SELECT AVG(x.price) FROM Magazine x");
Number result = (Number) q.getSingleResult ();
```

The following query will return the highest price of all the magazines titled "JDJ":

```
EntityManager em = ...
Query q = em.createQuery ("SELECT MAX(x.price) FROM Magazine x WHERE x.title = 'JDJ'");
Number result = (Number) q.getSingleResult ();
```

10.1.9. Named Queries

Query templates can be statically declared using the `NamedQuery` and `NamedQueries` annotations. For example:

```
@Entity
@NamedQueries({
    @NamedQuery(name="magsOverPrice",
        query="SELECT x FROM Magazine x WHERE x.price > ?1"),
    @NamedQuery(name="magsByTitle",
        query="SELECT x FROM Magazine x WHERE x.title = :titleParam")
})
public class Magazine
{
    ...
}
```

These declarations will define two named queries called `magsOverPrice` and `magsByTitle`.

```
public Query createNamedQuery (String name);
```

You retrieve named queries with the above `EntityManager` method. For example:

```
EntityManager em = ...
Query q = em.createNamedQuery ("magsOverPrice");
q.setParameter (1, 5.0f);
List<Magazine> results = q.getResultList ();
```

```
EntityManager em = ...
Query q = em.createNamedQuery ("magsByTitle");
q.setParameter ("titleParam", "JDJ");
List<Magazine> results = q.getResultList ();
```

10.1.10. Delete By Query

Queries are useful not only for finding objects, but for efficiently deleting them as well. For example, you might delete all records created before a certain date. Rather than bringing these objects into memory and delete them individually, JPA allows you to perform a single bulk delete based on JPQL criteria.

Delete by query uses the same JPQL syntax as normal queries, with one exception: begin your query string with the `delete` keyword instead of the `select` keyword. To then execute the delete, you call the following `Query` method:

```
public int executeUpdate ();
```

This method returns the number of objects deleted. The following example deletes all subscriptions whose expiration date has passed.

Example 10.1. Delete by Query

```
Query q = em.createQuery ("DELETE s FROM Subscription s WHERE s.subscriptionDate < :today");
q.setParameter ("today", new Date ());
int deleted = q.executeUpdate ();
```

10.1.11. Update By Query

Similar to bulk deletes, it is sometimes necessary to perform updates against a large number of queries in a single operation, without having to bring all the instances down to the client. Rather than bring these objects into memory and modifying them individually, JPA allows you to perform a single bulk update based on JPQL criteria.

Update by query uses the same JPQL syntax as normal queries, except that the query string begins with the `update` keyword instead of `select`. To execute the update, you call the following `Query` method:

```
public int executeUpdate ();
```

This method returns the number of objects updated. The following example updates all subscriptions whose expiration date has passed to have the "paid" field set to true.

Example 10.2. Update by Query

```
Query q = em.createQuery ("UPDATE Subscription s SET s.paid = :paid WHERE s.subscriptionDate < :today");
q.setParameter ("today", new Date ());
q.setParameter ("paid", true);
int updated = q.executeUpdate ();
```

10.2. JPQL Language Reference

The Java Persistence query language (JPQL) is used to define searches against persistent entities independent of the mechanism used to store those entities. As such, JPQL is "portable", and not constrained to any particular data store. The Java Persistence query language is an extension of the Enterprise JavaBeans query language, EJB QL, adding operations such as bulk deletes and updates, join operations, aggregates, projections, and subqueries. Furthermore, JPQL queries can be declared statically in metadata, or can be dynamically built in code. This chapter provides the full definition of the language.

Note

Much of this section is paraphrased or taken directly from Chapter 4 of the JSR 220 specification.

10.2.1. JPQL Statement Types

A JPQL statement may be either a `SELECT` statement, an `UPDATE` statement, or a `DELETE` statement. This chapter refers to all such statements as "queries". Where it is important to distinguish among statement types, the specific statement type is referenced. In BNF syntax, a query language statement is defined as:

- `QL_statement ::= select_statement | update_statement | delete_statement`

The complete BNF for JPQL is defined in [Section 10.2.12, “JPQL BNF” \[112\]](#) Any JPQL statement may be constructed dynamically or may be statically defined in a metadata annotation or XML descriptor element. All statement types may have parameters, as discussed in [Section 10.2.5.4, “JPQL Input Parameters” \[101\]](#)

10.2.1.1. JPQL Select Statement

A select statement is a string which consists of the following clauses:

- a `SELECT` clause, which determines the type of the objects or values to be selected;
- a `FROM` clause, which provides declarations that designate the domain to which the expressions specified in the other clauses of the query apply;
- an optional `WHERE` clause, which may be used to restrict the results that are returned by the query;
- an optional `GROUP BY` clause, which allows query results to be aggregated in terms of groups;
- an optional `HAVING` clause, which allows filtering over aggregated groups;
- an optional `ORDER BY` clause, which may be used to order the results that are returned by the query.

In BNF syntax, a select statement is defined as:

- `select_statement ::= select_clause from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]`

A select statement must always have a `SELECT` and a `FROM` clause. The square brackets `[]` indicate that the other clauses are optional.

10.2.1.2. JPQL Update and Delete Statements

Update and delete statements provide bulk operations over sets of entities. In BNF syntax, these operations are defined as:

- `update_statement ::= update_clause [where_clause]`
- `delete_statement ::= delete_clause [where_clause]`

The update and delete clauses determine the type of the entities to be updated or deleted. The `WHERE` clause may be used to restrict the scope of the update or delete operation. Update and delete statements are described further in [Section 10.2.9, “JPQL Bulk Update and Delete” \[111\]](#)

10.2.2. JPQL Abstract Schema Types and Query Domains

The Java Persistence query language is a typed language, and every expression has a type. The type of an expression is derived from the structure of the expression, the abstract schema types of the identification variable declarations, the types to which the persistent fields and relationships evaluate, and the types of literals. The abstract schema type of an entity is derived from the entity class and the metadata information provided by Java language annotations or in the XML descriptor.

Informally, the abstract schema type of an entity can be characterized as follows:

- For every persistent field or get accessor method (for a persistent property) of the entity class, there is a field ("state-field") whose abstract schema type corresponds to that of the field or the result type of the accessor method.
- For every persistent relationship field or get accessor method (for a persistent relationship property) of the entity class, there is a field ("association-field") whose type is the abstract schema type of the related entity (or, if the relationship is a one-to-many or many-to-many, a collection of such). Abstract schema types are specific to the query language data model. The persistence provider is not required to implement or otherwise materialize an abstract schema type. The domain of a query consists of the abstract schema types of all entities that are defined in the same persistence unit. The domain of a query may be restricted by the navigability of the relationships of the entity on which it is based. The association-fields of an entity's abstract schema type determine navigability. Using the association-fields and their values, a query can select related entities and use their abstract schema types in the query.

10.2.2.1. JPQL Entity Naming

Entities are designated in query strings by their entity names. The entity name is defined by the name element of the Entity annotation (or the entity-name XML descriptor element), and defaults to the unqualified name of the entity class. Entity names are scoped within the persistence unit and must be unique within the persistence unit.

10.2.2.2. JPQL Schema Example

This example assumes that the application developer provides several entity classes representing magazines, publishers, authors, and articles. The abstract schema types for these entities are `Magazine`, `Publisher`, `Author`, and `Article`.

The entity `Publisher` has a one-to-many relationships with `Magazine`. There is also a one-to-many relationship between `Magazine` and `Article`. The entity `Article` is related to `Author` in a one-to-one relationship.

Queries to select magazines can be defined by navigating over the association-fields and state-fields defined by `Magazine` and `Author`. A query to find all magazines that have unpublished articles is as follows:

```
SELECT DISTINCT mag FROM Magazine AS mag JOIN mag.articles AS art WHERE art.published = FALSE
```

This query navigates over the association-field `authors` of the abstract schema type `Magazine` to find articles, and uses the state-field `published` of `Article` to select those magazines that have at least one article that is published. Although predefined reserved identifiers, such as `DISTINCT`, `FROM`, `AS`, `JOIN`, `WHERE`, and `FALSE`, appear in upper case in this example, predefined reserved identifiers are case insensitive. The `SELECT` clause of this example designates the return type of this query to be of type `Magazine`. Because the same persistence unit defines the abstract persistence schemas of the related entities, the developer can also specify a query over `articles` that utilizes the abstract schema type for products, and hence the state-fields and association-fields of both the abstract schema types `Magazine` and `Author`. For example, if the abstract schema type `Author` has a state-field named `firstName`, a query over `articles` can be specified using this state-field. Such a query might be to find all magazines that have articles authored by someone with the first name "John".

```
SELECT DISTINCT mag FROM Magazine mag
JOIN mag.articles art JOIN art.author auth WHERE auth.firstName = 'John'
```

Because `Magazine` is related to `Author` by means of the relationships between `Magazine` and `Article` and between `Article` and `Author`, navigation using the association-fields `authors` and `product` is used to express the query. This query is specified by using the abstract schema name `Magazine`, which designates the abstract schema type over which the query ranges. The basis for the navigation is provided by the association-fields `authors` and `product` of the abstract schema types `Magazine` and `Article` respectively.

10.2.3. JPQL FROM Clause and Navigational Declarations

The `FROM` clause of a query defines the domain of the query by declaring identification variables. An identification variable is an identifier declared in the `FROM` clause of a query. The domain of the query may be constrained by path expressions. Identification variables designate instances of a particular entity abstract schema type. The `FROM` clause can contain multiple identification variable declarations separated by a comma (,).

- `from_clause ::= FROM identification_variable_declaration {, { identification_variable_declaration | collection_member_declaration } }*`
- `identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*`
- `range_variable_declaration ::= abstract_schema_name [AS] identification_variable`
- `join ::= join_spec join_association_path_expression [AS] identification_variable`
- `fetch_join ::= join_spec FETCH join_association_path_expression`
- `join_association_path_expression ::= join_collection_valued_path_expression | join_single_valued_association_path_expression`
- `join_spec ::= [LEFT [OUTER] | INNER] JOIN`
- `collection_member_declaration ::= IN (collection_valued_path_expression) [AS] identification_variable`

10.2.3.1. JPQL FROM Identifiers

An identifier is a character sequence of unlimited length. The character sequence must begin with a Java identifier start character, and all other characters must be Java identifier part characters. An identifier start character is any character for which the method `Character.isJavaIdentifierStart` returns `true`. This includes the underscore (`_`) character and the dollar-sign (`$`) character. An identifier-part character is any character for which the method `Character.isJavaIdentifierPart` returns `true`. The question-mark (`?`) character is reserved for use by the Java Persistence query language. The following are reserved identifiers:

- `SELECT`
- `FROM`
- `WHERE`
- `UPDATE`
- `DELETE`
- `JOIN`
- `OUTER`
- `INNER`
- `LEFT`
- `GROUP`
- `BY`

- HAVING
- FETCH
- DISTINCT
- OBJECT
- NULL
- TRUE
- FALSE
- NOT
- AND
- OR
- BETWEEN
- LIKE
- IN
- AS
- UNKNOWN
- EMPTY
- MEMBER
- OF
- IS
- AVG
- MAX
- MIN
- SUM
- COUNT
- ORDER
- BY
- ASC
- DESC
- MOD
- UPPER
- LOWER

- TRIM
- POSITION
- CHARACTER_LENGTH
- CHAR_LENGTH
- BIT_LENGTH
- CURRENT_TIME
- CURRENT_DATE
- CURRENT_TIMESTAMP
- NEW
- EXISTS
- ALL
- ANY
- SOME

Reserved identifiers are case insensitive. Reserved identifiers must not be used as identification variables. It is recommended that other SQL reserved words not be used as identification variables in queries, as they may be used as reserved identifiers in future releases of the specification.

10.2.3.2. JPQL Identification Variables

An identification variable is a valid identifier declared in the FROM clause of a query. All identification variables must be declared in the FROM clause. Identification variables cannot be declared in other clauses. An identification variable must not be a reserved identifier or have the same name as any entity in the same persistence unit: Identification variables are case insensitive. An identification variable evaluates to a value of the type of the expression used in declaring the variable. For example, consider the previous query:

```
SELECT DISTINCT mag FROM Magazine mag JOIN mag.articles art JOIN art.author auth WHERE auth.firstName = 'John'
```

In the FROM clause declaration `mag.articles art`, the identification variable `art` evaluates to any `Article` value directly reachable from `Magazine`. The association-field `articles` is a collection of instances of the abstract schema type `Article` and the identification variable `art` refers to an element of this collection. The type of `auth` is the abstract schema type of `Author`. An identification variable ranges over the abstract schema type of an entity. An identification variable designates an instance of an entity abstract schema type or an element of a collection of entity abstract schema type instances. Identification variables are existentially quantified in a query. An identification variable always designates a reference to a single value. It is declared in one of three ways: in a range variable declaration, in a join clause, or in a collection member declaration. The identification variable declarations are evaluated from left to right in the FROM clause, and an identification variable declaration can use the result of a preceding identification variable declaration of the query string.

10.2.3.3. JPQL Range Declarations

The syntax for declaring an identification variable as a range variable is similar to that of SQL; optionally, it uses the AS keyword.

- `range_variable_declaration ::= abstract_schema_name [AS] identification_variable`

Range variable declarations allow the developer to designate a "root" for objects which may not be reachable by navigation. In order to select values by comparing more than one instance of an entity abstract schema type, more than one identification variable ranging over the abstract schema type is needed in the FROM clause.

The following query returns magazines whose prices are greater than the price of magazines published by "Adventure" publishers. This example illustrates the use of two different identification variables in the FROM clause, both of the abstract schema type Magazine. The SELECT clause of this query determines that it is the magazines with prices greater than those of "Adventure" publisher's that are returned.

```
SELECT DISTINCT mag1 FROM Magazine mag1, Magazine mag2
WHERE mag1.price > mag2.price AND mag2.publisher.name = 'Adventure'
```

10.2.3.4. JPQL Path Expressions

An identification variable followed by the navigation operator (.) and a state-field or association-field is a path expression. The type of the path expression is the type computed as the result of navigation; that is, the type of the state-field or association-field to which the expression navigates. Depending on navigability, a path expression that leads to an association-field may be further composed. Path expressions can be composed from other path expressions if the original path expression evaluates to a single-valued type (not a collection) corresponding to an association-field. Path-expression navigability is composed using "inner join" semantics. That is, if the value of a non-terminal association-field in the path expression is null, the path is considered to have no value, and does not participate in the determination of the result. The syntax for single-valued path expressions and collection-valued path expressions is as follows:

- `single_valued_path_expression ::= state_field_path_expression | single_valued_association_path_expression`
- `state_field_path_expression ::= {identification_variable | single_valued_association_path_expression}.state_field`
- `single_valued_association_path_expression ::= identification_variable.{single_valued_association_field.*single_valued_association_field}`
- `collection_valued_path_expression ::= identification_variable.{single_valued_association_field.*collection_valued_association_field}`
- `state_field ::= {embedded_class_state_field.*simple_state_field}`

A `single_valued_association_field` is designated by the name of an association-field in a one-to-one or many-to-one relationship. The type of a `single_valued_association_field` and thus a `single_valued_association_path_expression` is the abstract schema type of the related entity. A `collection_valued_association_field` is designated by the name of an association-field in a one-to-many or a many-to-many relationship. The type of a `collection_valued_association_field` is a collection of values of the abstract schema type of the related entity. An `embedded_class_state_field` is designated by the name of an entity-state field that corresponds to an embedded class. Navigation to a related entity results in a value of the related entity's abstract schema type.

The evaluation of a path expression terminating in a state-field results in the abstract schema type corresponding to the Java type designated by the state-field. It is syntactically illegal to compose a path expression from a path expression that evaluates to a collection. For example, if `mag` designates `Magazine`, the path expression `mag.articles.author` is illegal since navigation to authors results in a collection. This case should produce an error when the query string is verified. To handle such a navigation, an identification variable must be declared in the FROM clause to range over the elements of the `articles` collection. Another path expression must be used to navigate over each such element in the WHERE clause of the query, as in the following query, which returns all authors that have any articles in any magazines:

```
SELECT DISTINCT art.author FROM Magazine AS mag, IN(mag.articles) art
```

10.2.3.5. JPQL Joins

An inner join may be implicitly specified by the use of a cartesian product in the FROM clause and a join condition in the WHERE clause.

The syntax for explicit join operations is as follows:

- `join ::= join_spec join_association_path_expression [AS] identification_variable`
- `fetch_join ::= join_spec FETCH join_association_path_expression`
- `join_association_path_expression ::= join_collection_valued_path_expression | join_single_valued_association_path_expression`
- `join_spec ::= [LEFT [OUTER] | INNER] JOIN`

The following inner and outer join operation types are supported.

10.2.3.5.1. JPQL Inner Joins (Relationship Joins)

The syntax for the inner join operation is

```
[ INNER ] JOIN join_association_path_expression [AS] identification_variable
```

For example, the query below joins over the relationship between publishers and magazines. This type of join typically equates to a join over a foreign key relationship in the database.

```
SELECT pub FROM Publisher pub JOIN pub.magazines mag WHERE pub.revenue > 1000000
```

The keyword INNER may optionally be used:

```
SELECT pub FROM Publisher pub INNER JOIN pub.magazines mag WHERE pub.revenue > 1000000
```

This is equivalent to the following query using the earlier IN construct. It selects those publishers with revenue of over 1 million for which at least one magazine exists:

```
SELECT OBJECT(pub) FROM Publisher pub, IN(pub.magazines) mag WHERE pub.revenue > 1000000
```

10.2.3.5.2. JPQL Outer Joins

LEFT JOIN and LEFT OUTER JOIN are synonymous. They enable the retrieval of a set of entities where matching values in the join condition may be absent. The syntax for a left outer join is:

```
LEFT [OUTER] JOIN join_association_path_expression [AS] identification_variable
```

For example:

```
SELECT pub FROM Publisher pub LEFT JOIN pub.magazines mag WHERE pub.revenue > 1000000
```

The keyword OUTER may optionally be used:

```
SELECT pub FROM Publisher pub LEFT OUTER JOIN pub.magazines mags WHERE pub.revenue > 1000000
```

An important use case for LEFT JOIN is in enabling the prefetching of related data items as a side effect of a query. This is accomplished by specifying the LEFT JOIN as a FETCH JOIN.

10.2.3.5.3. JPQL Fetch Joins

A FETCH JOIN enables the fetching of an association as a side effect of the execution of a query. A FETCH JOIN is specified over an entity and its related entities. The syntax for a fetch join is

- `fetch_join ::= [LEFT [OUTER] | INNER] JOIN FETCH join_association_path_expression`

The association referenced by the right side of the FETCH JOIN clause must be an association that belongs to an entity that is returned as a result of the query. It is not permitted to specify an identification variable for the entities referenced by the right side of the FETCH JOIN clause, and hence references to the implicitly fetched entities cannot appear elsewhere in the query. The following query returns a set of magazines. As a side effect, the associated articles for those magazines are also retrieved, even though they are not part of the explicit query result. The persistent fields or properties of the articles that are eagerly fetched are fully initialized. The initialization of the relationship properties of the `articles` that are retrieved is determined by the metadata for the `Article` entity class.

```
SELECT mag FROM Magazine mag LEFT JOIN FETCH mag.articles WHERE mag.id = 1
```

A fetch join has the same join semantics as the corresponding inner or outer join, except that the related objects specified on the right-hand side of the join operation are not returned in the query result or otherwise referenced in the query. Hence, for example, if magazine id 1 has five articles, the above query returns five references to the magazine 1 entity.

10.2.3.6. JPQL Collection Member Declarations

An identification variable declared by a `collection_member_declaration` ranges over values of a collection obtained by navigation using a path expression. Such a path expression represents a navigation involving the association-fields of an entity abstract schema type. Because a path expression can be based on another path expression, the navigation can use the association-fields of related entities. An identification variable of a collection member declaration is declared using a special operator, the reserved identifier `IN`. The argument to the `IN` operator is a collection-valued path expression. The path expression evaluates to a collection type specified as a result of navigation to a collection-valued association-field of an entity abstract schema type. The syntax for declaring a collection member identification variable is as follows:

- `collection_member_declaration ::= IN (collection_valued_path_expression) [AS] identification_variable`

For example, the query

```
SELECT DISTINCT mag FROM Magazine mag
      JOIN mag.articles art
      JOIN art.author auth
```

```
WHERE auth.lastName = 'Grisham'
```

may equivalently be expressed as follows, using the `IN` operator:

```
SELECT DISTINCT mag FROM Magazine mag,  
IN(mag.articles) art  
WHERE art.author.lastName = 'Grisham'
```

In this example, `articles` is the name of an association-field whose value is a collection of instances of the abstract schema type `Article`. The identification variable `art` designates a member of this collection, a single `Article` abstract schema type instance. In this example, `mag` is an identification variable of the abstract schema type `Magazine`.

10.2.3.7. JPQL Polymorphism

Java Persistence queries are automatically polymorphic. The `FROM` clause of a query designates not only instances of the specific entity classes to which the query explicitly refers but of subclasses as well. The instances returned by a query include instances of the subclasses that satisfy the query criteria.

10.2.4. JPQL WHERE Clause

The `WHERE` clause of a query consists of a conditional expression used to select objects or values that satisfy the expression. The `WHERE` clause restricts the result of a select statement or the scope of an update or delete operation. A `WHERE` clause is defined as follows:

- `where_clause ::= WHERE conditional_expression`

The `GROUP BY` construct enables the aggregation of values according to the properties of an entity class. The `HAVING` construct enables conditions to be specified that further restrict the query result as restrictions upon the groups. The syntax of the `HAVING` clause is as follows:

- `having_clause ::= HAVING conditional_expression`

The `GROUP BY` and `HAVING` constructs are further discussed in [Section 10.2.6, “JPQL GROUP BY, HAVING” \[107\]](#)

10.2.5. JPQL Conditional Expressions

The following sections describe the language constructs that can be used in a conditional expression of the `WHERE` clause or `HAVING` clause. State-fields that are mapped in serialized form or as `lob`s may not be portably used in conditional expressions.

Note

The implementation is not expected to perform such query operations involving such fields in memory rather than in the database.

10.2.5.1. JPQL Literals

A string literal is enclosed in single quotes--for example: `'literal'`. A string literal that includes a single quote is represented by two single quotes--for example: `'literal''s'`. String literals in queries, like Java String literals, use unicode character encoding. The use of Java escape notation is not supported in query string literals. Exact numeric literals support the use of Java integer literal syntax

as well as SQL exact numeric literal syntax. Approximate literals support the use Java floating point literal syntax as well as SQL approximate numeric literal syntax. Enum literals support the use of Java enum literal syntax. The enum class name must be specified. Appropriate suffixes may be used to indicate the specific type of a numeric literal in accordance with the Java Language Specification. The boolean literals are `TRUE` and `FALSE`. Although predefined reserved literals appear in upper case, they are case insensitive.

10.2.5.2. JPQL Identification Variables

All identification variables used in the `WHERE` or `HAVING` clause of a `SELECT` or `DELETE` statement must be declared in the `FROM` clause, as described in [Section 10.2.3.2, “JPQL Identification Variables” \[96\]](#). The identification variables used in the `WHERE` clause of an `UPDATE` statement must be declared in the `UPDATE` clause. Identification variables are existentially quantified in the `WHERE` and `HAVING` clause. This means that an identification variable represents a member of a collection or an instance of an entity's abstract schema type. An identification variable never designates a collection in its entirety.

10.2.5.3. JPQL Path Expressions

It is illegal to use a `collection_valued_path_expression` within a `WHERE` or `HAVING` clause as part of a conditional expression except in an `empty_collection_comparison_expression`, in a `collection_member_expression`, or as an argument to the `SIZE` operator.

10.2.5.4. JPQL Input Parameters

Either positional or named parameters may be used. Positional and named parameters may not be mixed in a single query. Input parameters can only be used in the `WHERE` clause or `HAVING` clause of a query.

Note that if an input parameter value is null, comparison operations or arithmetic operations involving the input parameter will return an unknown value. See [Section 10.2.10, “JPQL Null Values” \[111\]](#)

10.2.5.4.1. JPQL Positional Parameters

The following rules apply to positional parameters.

- Input parameters are designated by the question mark (?) prefix followed by an integer. For example: ?1.
- Input parameters are numbered starting from 1. Note that the same parameter can be used more than once in the query string and that the ordering of the use of parameters within the query string need not conform to the order of the positional parameters.

10.2.5.4.2. JPQL Named Parameters

A named parameter is an identifier that is prefixed by the ":" symbol. It follows the rules for identifiers defined in [Section 10.2.3.1, “JPQL FROM Identifiers” \[94\]](#). Named parameters are case sensitive.

Example:

```
SELECT pub FROM Publisher pub WHERE pub.revenue > :rev
```

10.2.5.5. JPQL Conditional Expression Composition

Conditional expressions are composed of other conditional expressions, comparison operations, logical operations, path expressions that evaluate to boolean values, boolean literals, and boolean input parameters. Arithmetic expressions can be used in comparison expressions. Arithmetic expressions are composed of other arithmetic expressions, arithmetic operations, path expres-

sions that evaluate to numeric values, numeric literals, and numeric input parameters. Arithmetic operations use numeric promotion. Standard bracketing () for ordering expression evaluation is supported. Conditional expressions are defined as follows:

- `conditional_expression ::= conditional_term | conditional_expression OR conditional_term`
- `conditional_term ::= conditional_factor | conditional_term AND conditional_factor`
- `conditional_factor ::= [NOT] conditional_primary`
- `conditional_primary ::= simple_cond_expression | (conditional_expression)`
- `simple_cond_expression ::= comparison_expression | between_expression | like_expression | in_expression | null_comparison_expression | empty_collection_comparison_expression | collection_member_expression | exists_expression`

Aggregate functions can only be used in conditional expressions in a HAVING clause. See [Section 10.2.6, “JPQL GROUP BY, HAVING” \[107\]](#)

10.2.5.6. JPQL Operators and Operator Precedence

The operators are listed below in order of decreasing precedence.

- Navigation operator (.)
- Arithmetic operators: +, - unary *, / multiplication and division +, - addition and subtraction
- Comparison operators : =, >, >=, <, <=, <> (not equal), [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
- Logical operators: NOT AND OR

The following sections describe other operators used in specific expressions.

10.2.5.7. JPQL Between Expressions

The syntax for the use of the comparison operator [NOT] BETWEEN in a conditional expression is as follows:

```
arithmetic_expression [NOT] BETWEEN arithmetic_expression AND arithmetic_expression | string_expression [NOT]
BETWEEN string_expression AND string_expression | datetime_expression [NOT] BETWEEN datetime_expression AND date-
time_expression
```

The BETWEEN expression

```
x BETWEEN y AND z
```

is semantically equivalent to:

```
y <= x AND x <= z
```

The rules for unknown and NULL values in comparison operations apply. See [Section 10.2.10, “JPQL Null Values” \[111\]](#) Examples are:

```
p.age BETWEEN 15 and 19
```

is equivalent to

```
p.age >= 15 AND p.age <= 19
```

```
p.age NOT BETWEEN 15 and 19
```

is equivalent to

```
p.age < 15 OR p.age > 19
```

10.2.5.8. JPQL In Expressions

The syntax for the use of the comparison operator [NOT] IN in a conditional expression is as follows:

- `in_expression ::= state_field_path_expression [NOT] IN (in_item {, in_item}* | subquery)`
- `in_item ::= literal | input_parameter`

The `state_field_path_expression` must have a string, numeric, or enum value. The literal and/or `input_parameter` values must be like the same abstract schema type of the `state_field_path_expression` in type. (See [Section 10.2.11, “JPQL Equality and Comparison Semantics”](#) [112])

The results of the subquery must be like the same abstract schema type of the `state_field_path_expression` in type. Subqueries are discussed in [Section 10.2.5.15, “JPQL Subqueries”](#) [106] Examples are:

```
o.country IN ('UK', 'US', 'France')
```

is true for UK and false for Peru, and is equivalent to the expression:

```
(o.country = 'UK') OR (o.country = 'US') OR (o.country = ' France')
```

In the following expression:

```
o.country NOT IN ('UK', 'US', 'France')
```

is false for UK and true for Peru, and is equivalent to the expression:

```
NOT ((o.country = 'UK') OR (o.country = 'US') OR (o.country = 'France'))
```

There must be at least one element in the comma separated list that defines the set of values for the IN expression. If the value of a `state_field_path_expression` in an IN or NOT IN expression is NULL or unknown, the value of the expression is unknown.

10.2.5.9. JPQL Like Expressions

The syntax for the use of the comparison operator [NOT] LIKE in a conditional expression is as follows:

```
string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]
```

The `string_expression` must have a string value. The `pattern_value` is a string literal or a string-valued input parameter in which an underscore (`_`) stands for any single character, a percent (`%`) character stands for any sequence of characters (including the empty sequence), and all other characters stand for themselves. The optional `escape_character` is a single-character string literal or a character-valued input parameter (i.e., `char` or `Character`) and is used to escape the special meaning of the underscore and percent characters in `pattern_value`. Examples are:

```
address.phone LIKE '12%3'
```

is true for '123' '12993' and false for '1234'

```
asentence.word LIKE '_l_se'
```

is true for 'lose' and false for 'loose'

```
aword.underscored LIKE '\_%' ESCAPE '\'
```

is true for '_foo' and false for 'bar'

```
address.phone NOT LIKE '12%3'
```

is false for '123' and '12993' and true for '1234' If the value of the `string_expression` or `pattern_value` is `NULL` or unknown, the value of the `LIKE` expression is unknown. If the `escape_character` is specified and is `NULL`, the value of the `LIKE` expression is unknown.

10.2.5.10. JPQL Null Comparison Expressions

The syntax for the use of the comparison operator `IS NULL` in a conditional expression is as follows:

```
{single_valued_path_expression | input_parameter} IS [NOT] NULL
```

A null comparison expression tests whether or not the single-valued path expression or input parameter is a `NULL` value.

10.2.5.11. JPQL Empty Collection Comparison Expressions

The syntax for the use of the comparison operator `IS EMPTY` in an `empty_collection_comparison_expression` is as follows:

```
collection_valued_path_expression IS [NOT] EMPTY
```

This expression tests whether or not the collection designated by the collection-valued path expression is empty (i.e., has no elements).

For example, the following query will return all magazines that don't have any articles at all:

```
SELECT mag FROM Magazine mag WHERE mag.articles IS EMPTY
```

If the value of the collection-valued path expression in an empty collection comparison expression is unknown, the value of the empty comparison expression is unknown.

10.2.5.12. JPQL Collection Member Expressions

The use of the comparison `collection_member_expression` is as follows: syntax for the operator `MEMBER OF` in an

- `collection_member_expression ::= entity_expression [NOT] MEMBER [OF] collection_valued_path_expression`
- `entity_expression ::= single_valued_association_path_expression | simple_entity_expression`
- `simple_entity_expression ::= identification_variable | input_parameter`

This expression tests whether the designated value is a member of the collection specified by the collection-valued path expression. If the collection valued path expression designates an empty collection, the value of the `MEMBER OF` expression is `FALSE` and the value of the `NOT MEMBER OF` expression is `TRUE`. Otherwise, if the value of the collection-valued path expression or single-valued association-field path expression in the collection member expression is `NULL` or unknown, the value of the collection member expression is unknown.

10.2.5.13. JPQL Exists Expressions

An `EXISTS` expression is a predicate that is true only if the result of the subquery consists of one or more values and that is false otherwise. The syntax of an exists expression is

- `exists_expression ::= [NOT] EXISTS (subquery)`

The use of the reserved word `OF` is optional in this expression.

Example:

```
SELECT DISTINCT auth FROM Author auth
WHERE EXISTS
  (SELECT spouseAuthor FROM Author spouseAuthor WHERE spouseAuthor = auth.spouse)
```

The result of this query consists of all authors whose spouse is also an author.

10.2.5.14. JPQL All or Any Expressions

An `ALL` conditional expression is a predicate that is true if the comparison operation is true for all values in the result of the subquery or the result of the subquery is empty. An `ALL` conditional expression is false if the result of the comparison is false for at least one row, and is unknown if neither true nor false. An `ANY` conditional expression is a predicate that is true if the comparison operation is true for some value in the result of the subquery. An `ANY` conditional expression is false if the result of the subquery is empty or if the comparison operation is false for every value in the result of the subquery, and is unknown if neither true nor false. The keyword `SOME` is synonymous with `ANY`. The comparison operators used with `ALL` or `ANY` conditional expressions are `=`, `<`, `<=`, `>`, `>=`, `<>`. The result of the subquery must be like that of the other argument to the comparison operator in type. See

Section 10.2.11, “JPQL Equality and Comparison Semantics” [112] The syntax of an ALL or ANY expression is specified as follows:

- `all_or_any_expression ::= { ALL | ANY | SOME } (subquery)`

The following example select the authors who make the highest salary for their magazine:

```
SELECT auth FROM Author auth
WHERE auth.salary >= ALL(SELECT a.salary FROM Author a WHERE a.magazine = auth.magazine)
```

10.2.5.15. JPQL Subqueries

Subqueries may be used in the WHERE or HAVING clause. The syntax for subqueries is as follows:

- `subquery ::= simple_select_clause subquery_from_clause [where_clause] [groupby_clause] [having_clause]`

Subqueries are restricted to the WHERE and HAVING clauses in this release. Support for subqueries in the FROM clause will be considered in a later release of the specification.

- `simple_select_clause ::= SELECT [DISTINCT] simple_select_expression`
- `subquery_from_clause ::= FROM subselect_identification_variable_declaration {, subselect_identification_variable_declaration}*`
- `subselect_identification_variable_declaration ::= identification_variable_declaration | association_path_expression [AS] identification_variable | collection_member_declaration`
- `simple_select_expression ::= single_valued_path_expression | aggregate_expression | identification_variable`

Examples:

```
SELECT DISTINCT auth FROM Author auth
WHERE EXISTS (SELECT spouseAuth FROM Author spouseAuth WHERE spouseAuth = auth.spouse)
```

```
SELECT mag FROM Magazine mag
WHERE (SELECT COUNT(art) FROM mag.articles art) > 10
```

Note that some contexts in which a subquery can be used require that the subquery be a scalar subquery (i.e., produce a single result). This is illustrated in the following example involving a numeric comparison operation.

```
SELECT goodPublisher FROM Publisher goodPublisher
WHERE goodPublisher.revenue < (SELECT AVG(p.revenue) FROM Publisher p)
```

10.2.5.16. JPQL Functional Expressions

The JPQL includes the following built-in functions, which may be used in the WHERE or HAVING clause of a query. If the value of any argument to a functional expression is null or unknown, the value of the functional expression is unknown.

10.2.5.16.1. JPQL String Functions

- `functions_returning_strings ::= CONCAT(string_primary, string_primary) | SUBSTRING(string_primary, simple_arithmetic_expression, simple_arithmetic_expression) | TRIM([[trim_specification] [trim_character] FROM] string_primary) | LOWER(string_primary) | UPPER(string_primary)`
- `trim_specification ::= LEADING | TRAILING | BOTH`
- `functions_returning_numerics ::= LENGTH(string_primary) | LOCATE(string_primary, string_primary[, simple_arithmetic_expression])`

The `CONCAT` function returns a string that is a concatenation of its arguments. The second and third arguments of the `SUBSTRING` function denote the starting position and length of the substring to be returned. These arguments are integers. The first position of a string is denoted by 1. The `SUBSTRING` function returns a string. The `TRIM` function trims the specified character from a string. If the character to be trimmed is not specified, it is assumed to be space (or blank). The optional `trim_character` is a single-character string literal or a character-valued input parameter (i.e., `char` or `Character`). If a trim specification is not provided, `BOTH` is assumed. The `TRIM` function returns the trimmed string. The `LOWER` and `UPPER` functions convert a string to lower and upper case, respectively. They return a string. The `LOCATE` function returns the position of a given string within a string, starting the search at a specified position. It returns the first position at which the string was found as an integer. The first argument is the string to be located; the second argument is the string to be searched; the optional third argument is an integer that represents the string position at which the search is started (by default, the beginning of the string to be searched). The first position in a string is denoted by 1. If the string is not found, 0 is returned. The `LENGTH` function returns the length of the string in characters as an integer.

10.2.5.16.2. JPQL Arithmetic Functions

- `functions_returning_numerics ::= ABS(simple_arithmetic_expression) | SQRT(simple_arithmetic_expression) | MOD(simple_arithmetic_expression, simple_arithmetic_expression) | SIZE(collection_valued_path_expression)`

The `ABS` function takes a numeric argument and returns a number (integer, float, or double) of the same type as the argument to the function. The `SQRT` function takes a numeric argument and returns a double.

Note that not all databases support the use of a trim character other than the space character; use of this argument may result in queries that are not portable. Note that not all databases support the use of the third argument to `LOCATE`; use of this argument may result in queries that are not portable.

The `MOD` function takes two integer arguments and returns an integer. The `SIZE` function returns an integer value, the number of elements of the collection. If the collection is empty, the `SIZE` function evaluates to zero. Numeric arguments to these functions may correspond to the numeric Java object types as well as the primitive numeric types.

10.2.5.16.3. JPQL Datetime Functions

`functions_returning_datetime ::= CURRENT_DATE | CURRENT_TIME | CURRENT_TIMESTAMP`

The datetime functions return the value of current date, time, and timestamp on the database server.

10.2.6. JPQL GROUP BY, HAVING

The `GROUP BY` construct enables the aggregation of values according to a set of properties. The `HAVING` construct enables con-

ditions to be specified that further restrict the query result. Such conditions are restrictions upon the groups. The syntax of the `GROUP BY` and `HAVING` clauses is as follows:

- `groupby_clause ::= GROUP BY groupby_item {, groupby_item}*`
- `groupby_item ::= single_valued_path_expression | identification_variable`
- `having_clause ::= HAVING conditional_expression`

If a query contains both a `WHERE` clause and a `GROUP BY` clause, the effect is that of first applying the where clause, and then forming the groups and filtering them according to the `HAVING` clause. The `HAVING` clause causes those groups to be retained that satisfy the condition of the `HAVING` clause. The requirements for the `SELECT` clause when `GROUP BY` is used follow those of SQL: namely, any item that appears in the `SELECT` clause (other than as an argument to an aggregate function) must also appear in the `GROUP BY` clause. In forming the groups, null values are treated as the same for grouping purposes. Grouping by an entity is permitted. In this case, the entity must contain no serialized state fields or lob-valued state fields. The `HAVING` clause must specify search conditions over the grouping items or aggregate functions that apply to grouping items.

If there is no `GROUP BY` clause and the `HAVING` clause is used, the result is treated as a single group, and the select list can only consist of aggregate functions. When a query declares a `HAVING` clause, it must always also declare a `GROUP BY` clause.

10.2.7. JPQL SELECT Clause

The `SELECT` clause denotes the query result. More than one value may be returned from the `SELECT` clause of a query. The `SELECT` clause may contain one or more of the following elements: a single range variable or identification variable that ranges over an entity abstract schema type, a single-valued path expression, an aggregate select expression, a constructor expression. The `SELECT` clause has the following syntax:

- `select_clause ::= SELECT [DISTINCT] select_expression {, select_expression}*`
- `select_expression ::= single_valued_path_expression | aggregate_expression | identification_variable | OBJECT(identification_variable) | constructor_expression`
- `constructor_expression ::= NEW constructor_name (constructor_item {, constructor_item}*)`
- `constructor_item ::= single_valued_path_expression | aggregate_expression`
- `aggregate_expression ::= { AVG | MAX | MIN | SUM } ([DISTINCT] state_field_path_expression) | COUNT ([DISTINCT] identification_variable | state_field_path_expression | single_valued_association_path_expression)`

For example:

```
SELECT pub.id, pub.revenue
FROM Publisher pub JOIN pub.magazines mag WHERE mag.price > 5.00
```

Note that the `SELECT` clause must be specified to return only single-valued expressions. The query below is therefore not valid:

```
SELECT mag.authors FROM Magazine AS mag
```

The `DISTINCT` keyword is used to specify that duplicate values must be eliminated from the query result. If `DISTINCT` is not specified, duplicate values are not eliminated. Standalone identification variables in the `SELECT` clause may optionally be quali-

fied by the OBJECT operator. The SELECT clause must not use the OBJECT operator to qualify path expressions.

10.2.7.1. JPQL Result Type of the SELECT Clause

The type of the query result specified by the SELECT clause of a query is an entity abstract schema type, a state-field type, the result of an aggregate function, the result of a construction operation, or some sequence of these. The result type of the SELECT clause is defined by the result types of the select_expressions contained in it. When multiple select_expressions are used in the SELECT clause, the result of the query is of type Object[], and the elements in this result correspond in order to the order of their specification in the SELECT clause and in type to the result types of each of the select_expressions. The type of the result of a select_expression is as follows:

- A single_valued_path_expression that is a state_field_path_expression results in an object of the same type as the corresponding state field of the entity. If the state field of the entity is a primitive type, the corresponding object type is returned.
- single_valued_path_expression that is a single_valued_association_path_expression results in an entity object of the type of the relationship field or the subtype of the relationship field of the entity object as determined by the object/relational mapping.
- The result type of an identification_variable is the type of the entity to which that identification variable corresponds or a subtype as determined by the object/relational mapping.
- The result type of aggregate_expression is defined in section **Section 10.2.7.4, “JPQL Aggregate Functions” [109]**
- The result type of a constructor_expression is the type of the class for which the constructor is defined. The types of the arguments to the constructor are defined by the above rules.

10.2.7.2. JPQL Constructor Expressions

in the SELECT Clause A constructor may be used in the SELECT list to return one or more Java instances. The specified class is not required to be an entity or to be mapped to the database. The constructor name must be fully qualified.

If an entity class name is specified in the SELECT NEW clause, the resulting entity instances are in the new state.

```
SELECT NEW com.company.PublisherInfo(pub.id, pub.revenue, mag.price)
FROM Publisher pub JOIN pub.magazines mag WHERE mag.price > 5.00
```

10.2.7.3. JPQL Null Values in the Query Result

If the result of a query corresponds to a association-field or state-field whose value is null, that null value is returned in the result of the query method. The IS NOT NULL construct can be used to eliminate such null values from the result set of the query. Note, however, that state-field types defined in terms of Java numeric primitive types cannot produce NULL values in the query result. A query that returns such a state-field type as a result type must not return a null value.

10.2.7.4. JPQL Aggregate Functions

in the SELECT Clause The result of a query may be the result of an aggregate function applied to a path expression. The following aggregate functions can be used in the SELECT clause of a query: AVG, COUNT, MAX, MIN, SUM. For all aggregate functions except COUNT, the path expression that is the argument to the aggregate function must terminate in a state-field. The path expression argument to COUNT may terminate in either a state-field or a association-field, or the argument to COUNT may be an identification variable. Arguments to the functions SUM and AVG must be numeric. Arguments to the functions MAX and MIN must correspond to orderable state-field types (i.e., numeric types, string types, character types, or date types). The Java type that is contained in the result of a query using an aggregate function is as follows:

- COUNT returns Long.
- MAX, MIN return the type of the state-field to which they are applied.
- AVG returns Double.
- SUM returns Long when applied to state-fields of integral types (other than BigInteger); Double when applied to state-fields of floating point types; BigInteger when applied to state-fields of type BigInteger; and BigDecimal when applied to state-fields of type BigDecimal. If SUM, AVG, MAX, or MIN is used, and there are no values to which the aggregate function can be applied, the result of the aggregate function is NULL. If COUNT is used, and there are no values to which COUNT can be applied, the result of the aggregate function is 0.

The argument to an aggregate function may be preceded by the keyword `DISTINCT` to specify that duplicate values are to be eliminated before the aggregate function is applied. Null values are eliminated before the aggregate function is applied, regardless of whether the keyword `DISTINCT` is specified.

10.2.7.4.1. JPQL Aggregate Examples

Examples The following query returns the average price of all magazines:

```
SELECT AVG(mag.price) FROM Magazine mag
```

The following query returns the sum total cost of all the prices from all the magazines published by 'Larry':

```
SELECT SUM(mag.price) FROM Publisher pub JOIN pub.magazines mag pub.firstName = 'Larry'
```

The following query returns the total number of magazines:

```
SELECT COUNT(mag) FROM Magazine mag
```

10.2.8. JPQL ORDER BY Clause

The `ORDER BY` clause allows the objects or values that are returned by the query to be ordered. The syntax of the `ORDER BY` clause is

- `orderby_clause ::= ORDER BY orderby_item {, orderby_item}*`
- `orderby_item ::= state_field_path_expression [ASC | DESC]`

It is legal to specify `DISTINCT` with `MAX` or `MIN`, but it does not affect the result.

When the `ORDER BY` clause is used in a query, each element of the `SELECT` clause of the query must be one of the following: an identification variable `x`, optionally denoted as `OBJECT(x)`, a `single_valued_association_path_expression`, or a `state_field_path_expression`. For example:

```
SELECT pub FROM Publisher pub JOIN pub.magazines mag ORDER BY o.revenue, o.name
```

If more than one `orderby_item` is specified, the left-to-right sequence of the `orderby_item` elements determines the precedence, whereby the leftmost `orderby_item` has highest precedence. The keyword `ASC` specifies that ascending ordering be used; the keyword `DESC` specifies that descending ordering be used. Ascending ordering is the default. SQL rules for the ordering of null values apply: that is, all null values must appear before all non-null values in the ordering or all null values must appear after all non-null values in the ordering, but it is not specified which. The ordering of the query result is preserved in the result of the query method if the `ORDER BY` clause is used.

10.2.9. JPQL Bulk Update and Delete

Operations Bulk update and delete operations apply to entities of a single entity class (together with its subclasses, if any). Only one entity abstract schema type may be specified in the `FROM` or `UPDATE` clause. The syntax of these operations is as follows:

- `update_statement ::= update_clause [where_clause]`
- `update_clause ::= UPDATE abstract_schema_name [[AS] identification_variable] SET update_item {, update_item}*`
- `update_item ::= [identification_variable.]{state_field | single_valued_association_field} = new_value`
- `new_value ::= simple_arithmetic_expression | string_primary | datetime_primary | boolean_primary | enum_primary | simple_entity_expression | NULL`
- `delete_statement ::= delete_clause [where_clause]`
- `delete_clause ::= DELETE FROM abstract_schema_name [[AS] identification_variable]`

The syntax of the `WHERE` clause is described in [Section 10.2.4, “JPQL WHERE Clause” \[100\]](#). A delete operation only applies to entities of the specified class and its subclasses. It does not cascade to related entities. The `new_value` specified for an update operation must be compatible in type with the state-field to which it is assigned. Bulk update maps directly to a database update operation, bypassing optimistic locking checks. Portable applications must manually update the value of the version column, if desired, and/or manually validate the value of the version column. The persistence context is not synchronized with the result of the bulk update or delete. Caution should be used when executing bulk update or delete operations because they may result in inconsistencies between the database and the entities in the active persistence context. In general, bulk update and delete operations should only be performed within a separate transaction or at the beginning of a transaction (before entities have been accessed whose state might be affected by such operations).

Examples:

```
DELETE FROM Publisher pub WHERE pub.revenue > 1000000.0
```

```
DELETE FROM Publisher pub WHERE pub.revenue = 0 AND pub.magazines IS EMPTY
```

```
UPDATE Publisher pub SET pub.status = 'outstanding'
WHERE pub.revenue < 1000000 AND 20 > (SELECT COUNT(mag) FROM pub.magazines mag)
```

10.2.10. JPQL Null Values

When the target of a reference does not exist in the database, its value is regarded as `NULL`. SQL 92 `NULL` semantics defines the evaluation of conditional expressions containing `NULL` values. The following is a brief description of these semantics:

- Comparison or arithmetic operations with a `NULL` value always yield an unknown value.
- Two `NULL` values are not considered to be equal, the comparison yields an unknown value.
- Comparison or arithmetic operations with an unknown value always yield an unknown value.
- The `IS NULL` and `IS NOT NULL` operators convert a `NULL` state-field or single-valued association-field value into the respective `TRUE` or `FALSE` value.

Note: The JPQL defines the empty string, `""`, as a string with 0 length, which is not equal to a `NULL` value. However, `NULL` values and empty strings may not always be distinguished when queries are mapped to some databases. Application developers should therefore not rely on the semantics of query comparisons involving the empty string and `NULL` value.

10.2.11. JPQL Equality and Comparison Semantics

Only the values of like types are permitted to be compared. A type is like another type if they correspond to the same Java language type, or if one is a primitive Java language type and the other is the wrapped Java class type equivalent (e.g., `int` and `Integer` are like types in this sense). There is one exception to this rule: it is valid to compare numeric values for which the rules of numeric promotion apply. Conditional expressions attempting to compare non-like type values are disallowed except for this numeric case. Note that the arithmetic operators and comparison operators are permitted to be applied to state-fields and input parameters of the wrapped Java class equivalents to the primitive numeric Java types. Two entities of the same abstract schema type are equal if and only if they have the same primary key value. Only equality/inequality comparisons over enums are required to be supported.

10.2.12. JPQL BNF

The following is the BNF for the Java Persistence query language, from section 4.14 of the JSR 220 specification.

- `QL_statement ::= select_statement | update_statement | delete_statement`
- `select_statement ::= select_clause from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]`
- `update_statement ::= update_clause [where_clause]`
- `delete_statement ::= delete_clause [where_clause]`
- `from_clause ::= FROM identification_variable_declaration {, {identification_variable_declaration | collection_member_declaration} }*`
- `identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*`
- `range_variable_declaration ::= abstract_schema_name [AS] identification_variable`
- `join ::= join_spec join_association_path_expression [AS] identification_variable`
- `fetch_join ::= join_spec FETCH join_association_path_expression`
- `association_path_expression ::= collection_valued_path_expression | single_valued_association_path_expression`
- `join_spec ::= [LEFT [OUTER]]INNER JOIN`
- `join_association_path_expression ::= join_collection_valued_path_expression | join_single_valued_association_path_expression`
- `join_collection_valued_path_expression ::= identification_variable.collection_valued_association_field`

- `join_single_valued_association_path_expression ::= identification_variable.single_valued_association_field`
- `collection_member_declaration ::= IN (collection_valued_path_expression) [AS] identification_variable`
- `single_valued_path_expression ::= state_field_path_expression | single_valued_association_path_expression`
- `state_field_path_expression ::= {identification_variable | single_valued_association_path_expression}.state_field`
- `single_valued_association_path_expression ::= identification_variable.{single_valued_association_field.*single_valued_association_field}`
- `collection_valued_path_expression ::= identification_variable.{single_valued_association_field.*collection_valued_association_field}`
- `state_field ::= {embedded_class_state_field.*simple_state_field}`
- `update_clause ::= UPDATE abstract_schema_name [[AS] identification_variable] SET update_item {, update_item}*`
- `update_item ::= [identification_variable.]{state_field | single_valued_association_field}= new_value`
- `new_value ::= simple_arithmetic_expression | string_primary | datetime_primary | boolean_primary | enum_primary | simple_entity_expression | NULL`
- `delete_clause ::= DELETE FROM abstract_schema_name [[AS] identification_variable]`
- `select_clause ::= SELECT [DISTINCT] select_expression {, select_expression}*`
- `select_expression ::= single_valued_path_expression | aggregate_expression | identification_variable | OBJECT(identification_variable) | constructor_expression`
- `constructor_expression ::= NEW constructor_name(constructor_item {, constructor_item}*)`
- `constructor_item ::= single_valued_path_expression | aggregate_expression`
- `aggregate_expression ::= {AVG |MAX |MIN |SUM}([DISTINCT] state_field_path_expression) | COUNT ([DISTINCT] identification_variable | state_field_path_expression | single_valued_association_path_expression)`
- `where_clause ::= WHERE conditional_expression`
- `groupby_clause ::= GROUP BY groupby_item {, groupby_item}*`
- `groupby_item ::= single_valued_path_expression | identification_variable`
- `having_clause ::= HAVING conditional_expression`
- `orderby_clause ::= ORDER BY orderby_item {, orderby_item}*`
- `orderby_item ::= state_field_path_expression [ASC |DESC]`
- `subquery ::= simple_select_clause subquery_from_clause [where_clause] [groupby_clause] [having_clause]`
- `subquery_from_clause ::= FROM subselect_identification_variable_declaration {, subselect_identification_variable_declaration}*`
- `subselect_identification_variable_declaration ::= identification_variable_declaration | association_path_expression [AS] identification_variable | collection_member_declaration`
- `simple_select_clause ::= SELECT [DISTINCT] simple_select_expression`
- `simple_select_expression ::= single_valued_path_expression | aggregate_expression | identification_variable`

- conditional_expression ::= conditional_term | conditional_expression OR conditional_term
- conditional_term ::= conditional_factor | conditional_term AND conditional_factor
- conditional_factor ::= [NOT] conditional_primary
- conditional_primary ::= simple_cond_expression |(conditional_expression)
- simple_cond_expression ::= comparison_expression | between_expression | like_expression | in_expression | null_comparison_expression | empty_collection_comparison_expression | collection_member_expression | exists_expression
- between_expression ::= arithmetic_expression [NOT] BETWEEN arithmetic_expression AND arithmetic_expression | string_expression [NOT] BETWEEN string_expression AND string_expression | datetime_expression [NOT] BETWEEN datetime_expression AND datetime_expression
- in_expression ::= state_field_path_expression [NOT] IN(in_item {, in_item}* | subquery)
- in_item ::= literal | input_parameter
- like_expression ::= string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]
- null_comparison_expression ::= {single_valued_path_expression | input_parameter} IS [NOT] NULL
- empty_collection_comparison_expression ::= collection_valued_path_expression IS [NOT] EMPTY
- collection_member_expression ::= entity_expression [NOT] MEMBER [OF] collection_valued_path_expression
- exists_expression ::= [NOT] EXISTS(subquery)
- all_or_any_expression ::= {ALL |ANY |SOME}(subquery)
- comparison_expression ::= string_expressioncomparison_operator{string_expression|all_or_any_expression}| boolean_expression {=|<>} {boolean_expression | all_or_any_expression} | enum_expression {=|<>} {enum_expression | all_or_any_expression} | datetime_expression comparison_operator {datetime_expression | all_or_any_expression} | entity_expression {= |<>} {entity_expression | all_or_any_expression} | arithmetic_expression comparison_operator {arithmetic_expression | all_or_any_expression}
- comparison_operator ::= = > >= < <= <>
- arithmetic_expression ::= simple_arithmetic_expression |(subquery)
- simple_arithmetic_expression ::= arithmetic_term | simple_arithmetic_expression {+ |- } arithmetic_term
- arithmetic_term ::= arithmetic_factor | arithmetic_term {* / } arithmetic_factor
- arithmetic_factor ::= [{+ |-}] arithmetic_primary
- arithmetic_primary ::= state_field_path_expression | numeric_literal | (simple_arithmetic_expression) | input_parameter | functions_returning_numerics | aggregate_expression
- string_expression ::= string_primary |(subquery)
- string_primary ::= state_field_path_expression | string_literal | input_parameter | functions_returning_strings | aggregate_expression
- datetime_expression ::= datetime_primary |(subquery)
- datetime_primary ::= state_field_path_expression | input_parameter | functions_returning_datetime | aggregate_expression
- boolean_expression ::= boolean_primary |(subquery)

- `boolean_primary ::= state_field_path_expression | boolean_literal | input_parameter |`
- `enum_expression ::= enum_primary |(subquery)`
- `enum_primary ::= state_field_path_expression | enum_literal | input_parameter |`
- `entity_expression ::= single_valued_association_path_expression | simple_entity_expression`
- `simple_entity_expression ::= identification_variable | input_parameter`
- `functions_returning_numerics ::= LENGTH(string_primary)| LOCATE(string_primary,string_primary [, simple_arithmetic_expression]) | ABS(simple_arithmetic_expression) | SQRT(simple_arithmetic_expression) | MOD(simple_arithmetic_expression, simple_arithmetic_expression) | SIZE(collection_valued_path_expression)`
- `functions_returning_datetime ::= CURRENT_DATE| CURRENT_TIME | CURRENT_TIMESTAMP`
- `functions_returning_strings ::= CONCAT(string_primary, string_primary) | SUBSTRING(string_primary, simple_arithmetic_expression,simple_arithmetic_expression)| TRIM([[trim_specification] [trim_character] FROM] string_primary) | LOWER(string_primary) | UPPER(string_primary)`
- `trim_specification ::= LEADING | TRAILING | BOTH`

Chapter 11. SQL Queries

JPQL is a powerful query language, but there are times when it is not enough. Maybe you're migrating a JDBC application to JPA on a strict deadline, and you don't have time to translate your existing SQL selects to JPQL. Or maybe a certain query requires database-specific SQL your JPA implementation doesn't support. Or maybe your DBA has spent hours crafting the perfect select statement for a query in your application's critical path. Whatever the reason, SQL queries can remain an essential part of an application.

You are probably familiar with executing SQL queries by obtaining a `java.sql.Connection`, using the JDBC APIs to create a `Statement`, and executing that `Statement` to obtain a `ResultSet`. And of course, you are free to continue using this low-level approach to SQL execution in your JPA applications. However, JPA also supports executing SQL queries through the `javax.persistence.Query` interface introduced in [Chapter 10, JPA Query \[82\]](#). Using a JPA SQL query, you can retrieve either persistent objects or projections of column values. The following sections detail each use.

Important

This feature requires **Kodo Enterprise Edition**.

11.1. Creating SQL Queries

The `EntityManager` has two factory methods suitable for creating SQL queries:

```
public Query createNativeQuery (String sqlString, Class resultClass);
public Query createNativeQuery (String sqlString, String resultSetMapping);
```

The first method is used to create a new `Query` instance that will return instances of the specified class.

The second method uses a `SqlResultSetMapping` to determine the type of object or objects to return. The example below shows these methods in action.

Example 11.1. Creating a SQL Query

```
EntityManager em = ...;
Query query = em.createNativeQuery ("SELECT * FROM MAG", Magazine.class);
processMagazines (query.getResultList ());
```

Note

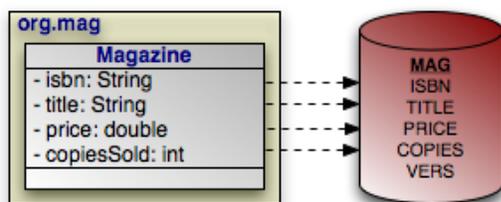
In addition to `SELECT` statements, Kodo supports stored procedure invocations as SQL queries. Kodo will assume any SQL that does not begin with the `SELECT` keyword (ignoring case) is a stored procedure call, and invoke it as such at the JDBC level.

11.2. Retrieving Persistent Objects with SQL

When you give a SQL Query a candidate class, it will return persistent instances of that class. At a minimum, your SQL must select the class' primary key columns, discriminator column (if mapped), and version column (also if mapped). The JPA runtime uses the values of the primary key columns to construct each result object's identity, and possibly to match it with a persistent object already in the EntityManager's cache. When an object is not already cached, the implementation creates a new object to represent the current result row. It might use the discriminator column value to make sure it constructs an object of the correct subclass. Finally, the query records available version column data for use in optimistic concurrency checking, should you later change the result object and flush it back to the database.

Aside from the primary key, discriminator, and version columns, any columns you select are used to populate the persistent fields of each result object. JPA implementations will compete on how effectively they map your selected data to your persistent instance fields.

Let's make the discussion above concrete with an example. It uses the following simple mapping between a class and the database:



Example 11.2. Retrieving Persistent Objects

```
Query query = em.createNativeQuery ("SELECT ISBN, TITLE, PRICE, "
    + "VERS FROM MAG WHERE PRICE > 5 AND PRICE < 10", Magazine.class);
List<Magazine> results = query.getResultList ();
for (Magazine mag : results)
    processMagazine (mag);
```

The query above works as advertised, but isn't very flexible. Let's update it to take in parameters for the minimum and maximum price, so we can reuse it to find magazines in any price range:

Example 11.3. SQL Query Parameters

```
Query query = em.createNativeQuery ("SELECT ISBN, TITLE, PRICE, "
    + "VERS FROM MAG WHERE PRICE > ?1 AND PRICE < ?2", Magazine.class);

query.setParameter (1, 5d);
query.setParameter (2, 10d);

List<Magazine> results = query.getResultList ();
for (Magazine mag : results)
    processMagazine (mag);
```

Like JDBC prepared statements, SQL queries represent parameters with question marks, but are followed by an integer to represent its index.

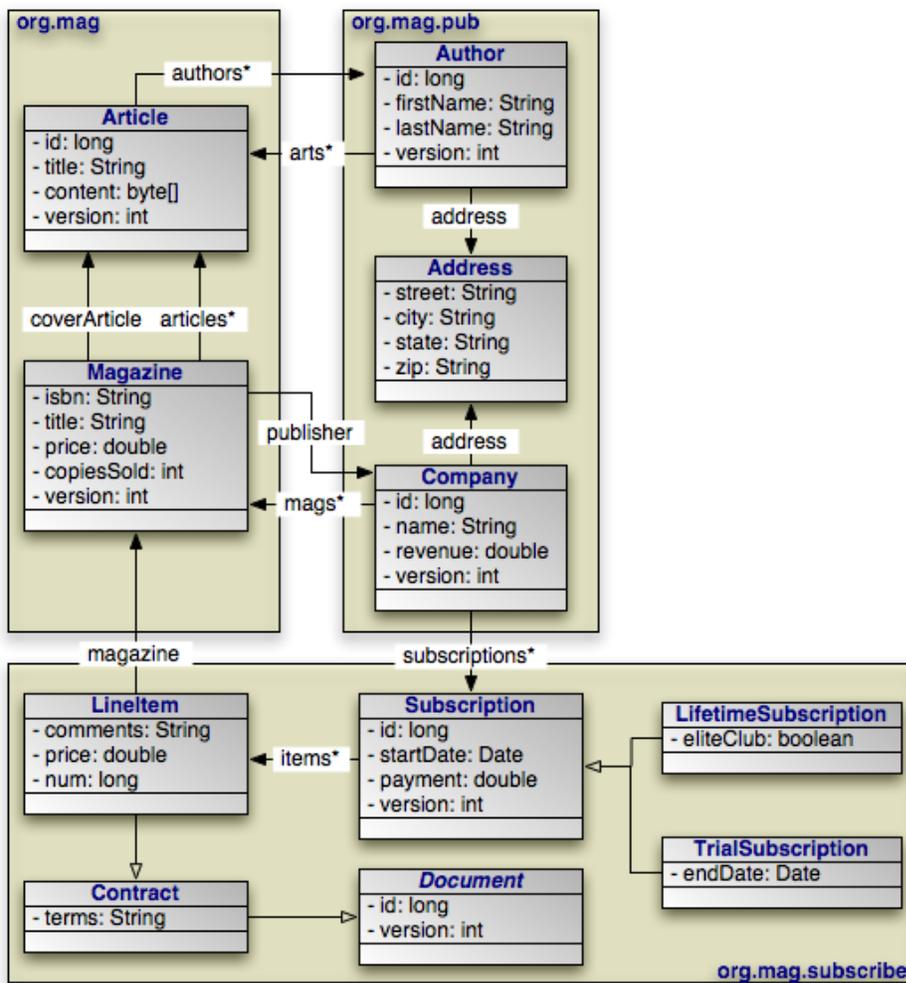
Chapter 12. Mapping Metadata

Object-relational mapping is the process of mapping entities to relational database tables. In EJB persistence, you perform object-relational mapping through *mapping metadata*. Mapping metadata uses annotations to describe how to link your object model to your relational model.

Note

Kodo offers tools to automate mapping and schema creation. See [Chapter 7, Mapping \[489\]](#) the Reference Guide.

Throughout this chapter, we will draw on the object model introduced in [Chapter 5, Metadata \[28\]](#). We present that model again below. As we discuss various aspects of mapping metadata, we will zoom in on specific areas of the model and show how we map the object layer to the relational layer.



All mapping metadata is optional. Where no explicit mapping metadata is given, EJB 3 persistence uses the defaults defined by the specification. As we present each mapping throughout this chapter, we also describe the defaults that apply when the mapping is absent.

12.1. Table

The `Table` annotation specifies the table for an entity class. If you omit the `Table` annotation, base entity classes default to a table with their unqualified class name. The default table of an entity subclass depends on the inheritance strategy, as you will see in [Section 12.6, “Inheritance” \[129\]](#)

Tables have the following properties:

- `String name`: The name of the table. Defaults to the unqualified entity class name.
- `String schema`: The table's schema. If you do not name a schema, EJB uses the default schema for the database connection.
- `String catalog`: The table's catalog. If you do not name a catalog, EJB uses the default catalog for the database connection.
- `UniqueConstraint[] uniqueConstraints`: An array of unique constraints to place on the table. We cover unique constraints below. Defaults to an empty array.

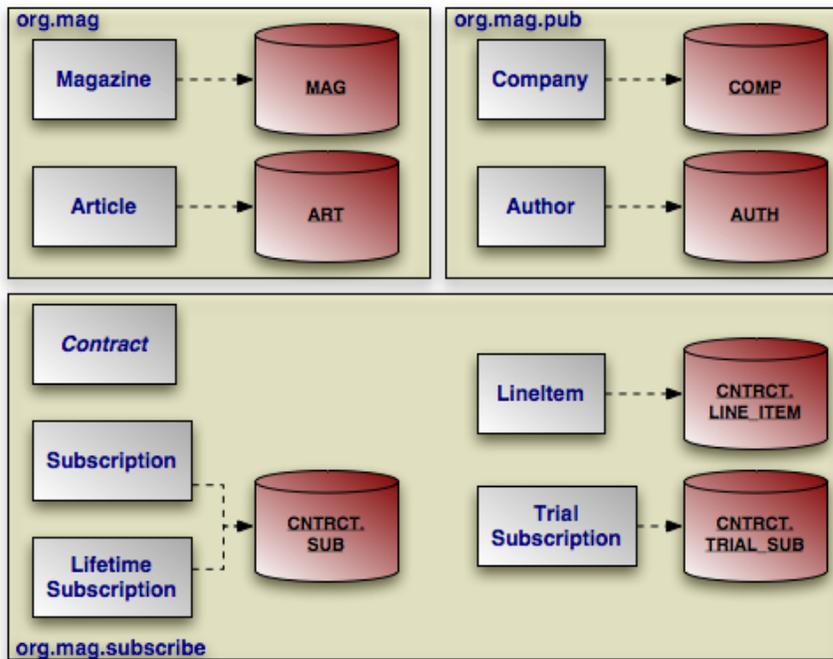
The equivalent XML element is `table`. It has the following attributes, which correspond to the annotation properties above:

- `name`
- `schema`
- `catalog`

The `table` element also accepts nested `unique-constraint` elements representing unique constraints. We will detail unique constraints shortly.

Sometimes, some of the fields in a class are mapped to secondary tables. In that case, use the class' `Table` annotation to name what you consider the class' primary table. Later, we will see how to map certain fields to other tables.

The example below maps classes to tables according to the following diagram. The `CONTRACT`, `SUB`, and `LINE_ITEM` tables are in the `CNTRCT` schema; all other tables are in the default schema.



Note that the diagram does not include our model's Document and Address classes. Mapped superclasses and embeddable classes are never mapped to tables.

Example 12.1. Mapping Classes

```

package org.mag;

@Entity
@IdClass(Magazine.MagazineId.class)
@Table(name="MAG")
public class Magazine
{
    ...

    public static class MagazineId
    {
        ...
    }
}

@Entity
@Table(name="ART")
public class Article
{
    ...
}

package org.mag.pub;

@Entity
@Table(name="COMP")
public class Company
{
    ...
}

@Entity
@Table(name="AUTH")
public class Author
{
    ...
}

@Embeddable
public class Address
{

```

```

}    ...

package org.mag.subscribe;

@MappedSuperclass
public abstract class Document
{
    ...
}

@Entity
@Table(schema="CNTRCT")
public class Contract
    extends Document
{
    ...
}

@Entity
@Table(name="SUB", schema="CNTRCT")
public class Subscription
{
    ...

    @Entity
    @Table(name="LINE_ITEM", schema="CNTRCT")
    public static class LineItem
        extends Contract
    {
        ...
    }
}

@Entity(name="Lifetime")
public class LifetimeSubscription
    extends Subscription
{
    ...
}

@Entity(name="Trial")
public class TrialSubscription
    extends Subscription
{
    ...
}

```

The same mapping information expressed in XML:

```

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0">
  <mapped-superclass class="org.mag.subscribe.Document">
    ...
  </mapped-superclass>
  <entity class="org.mag.Magazine">
    <table name="MAG"/>
    <id-class="org.mag.Magazine.MagazineId"/>
    ...
  </entity>
  <entity class="org.mag.Article">
    <table name="ART"/>
    ...
  </entity>
  <entity class="org.mag.pub.Company">
    <table name="COMP"/>
    ...
  </entity>
  <entity class="org.mag.pub.Author">
    <table name="AUTH"/>
    ...
  </entity>
  <entity class="org.mag.subscribe.Contract">
    <table schema="CNTRCT"/>
    ...
  </entity>
  <entity class="org.mag.subscribe.Subscription">
    <table name="SUB" schema="CNTRCT"/>
    ...
  </entity>
  <entity class="org.mag.subscribe.Subscription.LineItem">
    <table name="LINE_ITEM" schema="CNTRCT"/>
    ...
  </entity>

```

```

</entity>
<entity class="org.mag.subscribe.LifetimeSubscription" name="Lifetime">
  ...
</entity>
<entity class="org.mag.subscribe.TrialSubscription" name="Trial">
  ...
</entity>
<embeddable class="org.mag.pub.Address">
  ...
</embeddable>
</entity-mappings>

```

12.2. Unique Constraints

Unique constraints ensure that the data in a column or combination of columns is unique for each row. A table's primary key, for example, functions as an implicit unique constraint. In EJB persistence, you represent other unique constraints with an array of `UniqueConstraint` annotations within the table annotation. The unique constraints you define are used during table creation to generate the proper database constraints, and may also be used at runtime to order `INSERT`, `UPDATE`, and `DELETE` statements. For example, suppose there is a unique constraint on the columns of field `F`. In the same transaction, you remove an object `A` and persist a new object `B`, both with the same `F` value. The EJB persistence runtime must ensure that the SQL deleting `A` is sent to the database before the SQL inserting `B` to avoid a unique constraint violation.

`UniqueConstraint` has a single property:

- `String[] columnNames`: The names of the columns the constraint spans.

In XML, unique constraints are represented by nesting `unique-constraint` elements within the `table` element. Each `unique-constraint` element in turn nests `column-name` text elements to enumerate the constraint's columns.

Example 12.2. Defining a Unique Constraint

The following defines a unique constraint on the `TITLE` column of the `ART` table:

```

@Entity
@Table(name="ART", uniqueConstraints=@Unique(columnNames="TITLE"))
public class Article
{
  ...
}

```

The same metadata expressed in XML form:

```

<entity class="org.mag.Article">
  <table name="ART">
    <unique-constraint>
      <column-name>TITLE</column-name>
    </unique-constraint>
  </table>
  ...
</entity>

```

12.3. Column

In the previous section, we saw that a `UniqueConstraint` uses an array of column names. Field mappings, however, use full-fledged `Column` annotations. Column annotations have the following properties:

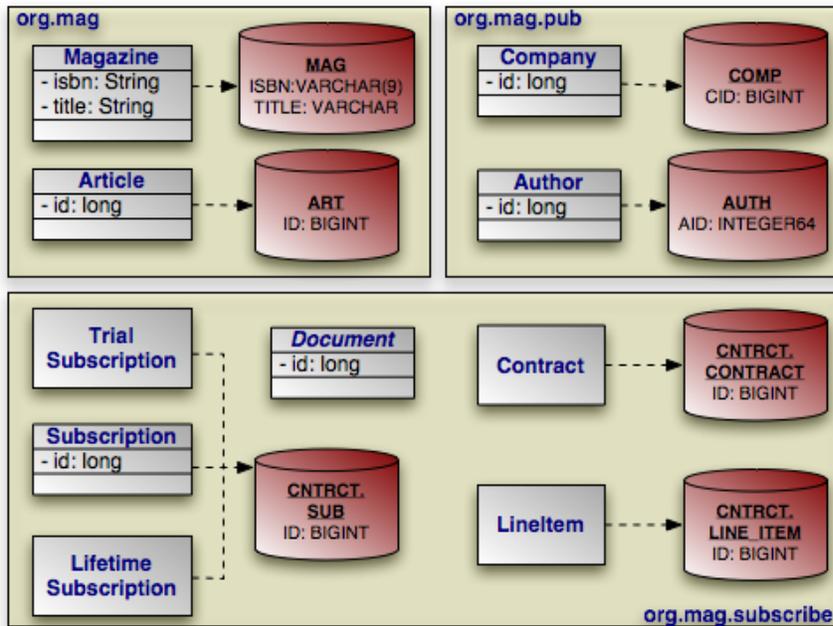
- `String name`: The column name. Defaults to the field name.
- `String columnDefinition`: The database-specific column type name. This property is only used by vendors that support creating tables from your mapping metadata. During table creation, the vendor will use the value of the `columnDefinition` as the declared column type. If no `columnDefinition` is given, the vendor will choose an appropriate default based on the field type combined with the column's length, precision, and scale.
- `int length`: The column length. This property is typically only used during table creation, though some vendors might use it to validate data before flushing. `CHAR` and `VARCHAR` columns typically default to a length of 255; other column types use the database default.
- `int precision`: The precision of a numeric column. This property is often used in conjunction with `scale` to form the proper column type name during table creation.
- `int scale`: The number of decimal digits a numeric column can hold. This property is often used in conjunction with `precision` to form the proper column type name during table creation.
- `boolean nullable`: Whether the column can store null values. Vendors may use this property both for table creation and at runtime; however, it is never required. Defaults to `true`.
- `boolean insertable`: By setting this property to `false`, you can omit the column from SQL `INSERT` statements. Defaults to `true`.
- `boolean updatable`: By setting this property to `false`, you can omit the column from SQL `UPDATE` statements. Defaults to `true`.
- `String table`: Sometimes you will need to map fields to tables other than the primary table. This property allows you specify that the column resides in a secondary table. We will see how to map fields to secondary tables later in the chapter.

The equivalent XML element is `column`. This element has attributes that are exactly equivalent to the `Column` annotation's properties described above:

- `name`
- `column-definition`
- `length`
- `precision`
- `scale`
- `insertable`
- `updatable`
- `table`

12.4. Identity Mapping

With our new knowledge of columns, we can map the identity fields of our entities. The diagram below now includes primary key columns for our model's tables. The primary key column for `Author` uses nonstandard type `INTEGER64`, and the `Magazine` . `isbn` field is mapped to a `VARCHAR(9)` column instead of a `VARCHAR(255)` column, which is the default for string fields. We do not need to point out either one of these oddities to the EJB persistence implementation for runtime use. If, however, we want to use the EJB persistence implementation to create our tables for us, it needs to know about any desired non-default column types. Therefore, the example following the diagram includes this data in its encoding of our mappings.



Note that many of our identity fields do not need to specify column information, because they use the default column name and type.

Example 12.3. Identity Mapping

```

package org.mag;

@Entity
@IdClass(Magazine.MagazineId.class)
@Table(name="MAG")
public class Magazine
{
    @Column(length=9)
    @Id private String isbn;
    @Id private String title;

    ...

    public static class MagazineId
    {
        ...
    }
}

@Entity
@Table(name="ART", uniqueConstraints=@Unique(columnNames="TITLE"))
public class Article
{
    @Id private long id;

    ...
}

package org.mag.pub;

@Entity
@Table(name="COMP")

```

```

public class Company
{
    @Column(name="CID")
    @Id private long id;

    ...
}

@Entity
@Table(name="AUTH")
public class Author
{
    @Column(name="AID", columnDefinition="INTEGER64")
    @Id private long id;

    ...
}

@Embeddable
public class Address
{
    ...
}

package org.mag.subscribe;

@MappedSuperclass
public abstract class Document
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

    ...
}

@Entity
@Table(schema="CNTRCT")
public class Contract
    extends Document
{
    ...
}

@Entity
@Table(name="SUB", schema="CNTRCT")
public class Subscription
{
    @Id private long id;

    ...

    @Entity
    @Table(name="LINE_ITEM", schema="CNTRCT")
    public static class LineItem
        extends Contract
    {
        ...
    }
}

@Entity(name="Lifetime")
public class LifetimeSubscription
    extends Subscription
{
    ...
}

@Entity(name="Trial")
public class TrialSubscription
    extends Subscription
{
    ...
}

```

The same metadata for Magazine and Company expressed in XML form:

```

<entity class="org.mag.Magazine">
  <id-class class="org.mag.Magazine.Magazine.MagazineId"/>
  <table name="MAG"/>
  <attributes>
    <id name="isbn">
      <column length="9"/>
    </id>
    <id name="title"/>
  </attributes>
</entity>

```

```

    ...
  </attributes>
</entity>
<entity class="org.mag.pub.Company">
  <table name="COMP"/>
  <attributes>
    <id name="id">
      <column name="CID"/>
    </id>
    ...
  </attributes>
</entity>

```

12.5. Generators

One aspect of identity mapping not covered in the previous section is EJB's ability to automatically assign a value to your numeric identity fields using *generators*. We discussed the available generator types in [Section 5.2.2, “Id” \[34\]](#). Now we show you how to define named generators.

12.5.1. Sequence Generator

Most databases allow you to create native sequences. These are database structures that generate increasing numeric values. The `SequenceGenerator` annotation represents a named database sequence. You can place the annotation on any package, entity class, persistent field declaration (if your entity uses field access), or getter method for a persistent property (if your entity uses property access). `SequenceGenerator` has the following properties:

- `String name`: The generator name. This property is required.
- `String sequenceName`: The name of the database sequence. If you do not specify the database sequence, your vendor will choose an appropriate default.
- `int initialValue`: The initial sequence value.
- `int allocationSize`: Some databases can pre-allocate groups of sequence values. This allows the database to service sequence requests from cache, rather than physically incrementing the sequence with every request. This allocation size defaults to 50.

Note

Kodo allows you to use describe one of Kodo's built-in generator implementations in the `sequenceName` property. You can also set the `sequenceName` to `system` to use the system sequence defined by the `kodo.Sequence` configuration property. See the Reference Guide's [Section 9.7, “Generators” \[555\]](#) for details.

The XML element for a sequence generator is `sequence-generator`. Its attributes mirror the above annotation's properties:

- `name`
- `sequence-name`
- `initial-value`
- `allocation-size`

To use a sequence generator, set your `GeneratedValue` annotation's `strategy` property to `GenerationType.SEQUENCE`, and its `generator` property to the sequence generator's declared name. Or equivalently, set your `generated-value` XML element's `strategy` attribute to `SEQUENCE` and its `generator` attribute to the generator name.

12.5.2. TableGenerator

A `TableGenerator` refers to a database table used to store increasing sequence values for one or more entities. As with `SequenceGenerator`, you can place the `TableGenerator` annotation on any package, entity class, persistent field declaration (if your entity uses field access), or getter method for a persistent property (if your entity uses property access). `TableGenerator` has the following properties:

- `String name`: The generator name. This property is required.
- `String table`: The name of the generator table. If left unspecified, your vendor will choose a default table.
- `String schema`: The named table's schema.
- `String catalog`: The named table's catalog.
- `String pkColumnName`: The name of the primary key column in the generator table. If unspecified, your implementation will choose a default.
- `String valueColumnName`: The name of the column that holds the sequence value. If unspecified, your implementation will choose a default.
- `String pkColumnValue`: The primary key column value of the row in the generator table holding this sequence value. You can use the same generator table for multiple logical sequences by supplying different `pkColumnValues`. If you do not specify a value, the implementation will supply a default.
- `int initialValue`: The value of the generator's first issued number.
- `int allocationSize`: The number of values to allocate in memory for each trip to the database. Allocating values in memory allows the EJB persistence runtime to avoid accessing the database for every sequence request. This number also specifies the amount that the sequence value is incremented each time the generator table is updated. Defaults to 50.

The XML equivalent is the `table-generator` element. This element's attributes correspond exactly to the above annotation's properties:

- `name`
- `table`
- `schema`
- `catalog`
- `pk-column-name`
- `value-column-name`
- `pk-column-value`
- `initial-value`
- `allocation-size`

To use a table generator, set your `GeneratedValue` annotation's `strategy` property to `GenerationType.TABLE`, and its `generator` property to the table generator's declared name. Or equivalently, set your `generated-value` XML element's `strategy` attribute to `TABLE` and its `generator` attribute to the generator name.

12.5.3. Example

Let's take advantage of generators in our entity model. Here are our updated mappings.

Example 12.4. Generator Mapping

```
package org.mag;

@Entity
@IdClass(Magazine.MagazineId.class)
@Table(name="MAG")
public class Magazine
{
    @Column(length=9)
    @Id private String isbn;
    @Id private String title;

    ...

    public static class MagazineId
    {
        ...
    }
}

@Entity
@Table(name="ART", uniqueConstraints=@Unique(columnNames="TITLE"))
@SequenceGenerator(name="ArticleSeq", sequenceName="ART_SEQ")
public class Article
{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="ArticleSeq")
    private long id;

    ...
}

package org.mag.pub;

@Entity
@Table(name="COMP")
public class Company
{
    @Column(name="CID")
    @Id private long id;

    ...
}

@Entity
@Table(name="AUTH")
public class Author
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="AuthorGen")
    @TableGenerator(name="AuthorGen", table="AUTH_GEN", pkColumnName="PK",
        valueColumnName="AID")
    @Column(name="AID", columnDefinition="INTEGER64")
    private long id;

    ...
}

@Embeddable
public class Address
{
    ...
}

package org.mag.subscribe;

@MappedSuperclass
public abstract class Document
{
    @Id
    @GeneratedValue(generator=GenerationType.IDENTITY)
```

```

    private long id;
    ...
}
@Entity
@Table(schema="CNTRCT")
public class Contract
    extends Document
{
    ...
}
@Entity
@Table(name="SUB", schema="CNTRCT")
public class Subscription
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;
    ...
    @Entity
    @Table(name="LINE_ITEM", schema="CNTRCT")
    public static class LineItem
        extends Contract
    {
        ...
    }
}
@Entity(name="Lifetime")
public class LifetimeSubscription
    extends Subscription
{
    ...
}
@Entity(name="Trial")
public class TrialSubscription
    extends Subscription
{
    ...
}

```

The same metadata for Article and Author expressed in XML form:

```

<entity class="org.mag.Article">
  <table name="ART">
    <unique-constraint>
      <column-name>TITLE</column-name>
    </unique-constraint>
  </table>
  <sequence-generator name="ArticleSeq" sequence-name="ART_SEQ"/>
  <attributes>
    <id name="id">
      <generated-value strategy="SEQUENCE" generator="ArticleSeq"/>
    </id>
    ...
  </attributes>
</entity>
<entity class="org.mag.pub.Author">
  <table name="AUTH"/>
  <attributes>
    <id name="id">
      <column name="AID" column-definition="INTEGER64"/>
      <generated-value strategy="TABLE" generator="AuthorGen"/>
      <table-generator name="AuthorGen" table="AUTH_GEN"
        pk-column-name="PK" value-column-name="AID"/>
    </id>
    ...
  </attributes>
</entity>

```

12.6. Inheritance

In the 1990's programmers coined the term *impedance mismatch* to describe the difficulties in bridging the object and relational worlds. Perhaps no feature of object modeling highlights the impedance mismatch better than inheritance. There is no natural, efficient way to represent an inheritance relationship in a relational database.

Luckily, EJB persistence gives you a choice of inheritance strategies, making the best of a bad situation. The base entity class defines the inheritance strategy for the hierarchy with the `Inheritance` annotation. `Inheritance` has the following properties:

- `InheritanceType` `strategy`: Enum value declaring the inheritance strategy for the hierarchy. Defaults to `InheritanceType.SINGLE_TABLE`. We detail each of the available strategies below.

The corresponding XML element is `inheritance`, which has a single attribute:

- `strategy`: One of `SINGLE_TABLE`, `JOINED`, or `TABLE_PER_CLASS`.

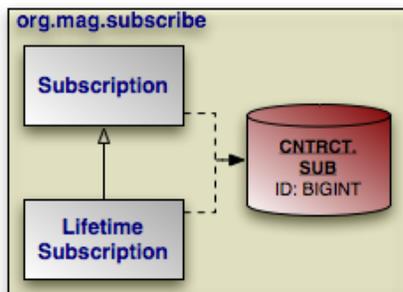
The following sections describe EJB's standard inheritance strategies.

Note

Kodo allows you to vary your inheritance strategy for each class, rather than forcing a single strategy per inheritance hierarchy. See [Section 7.7, “Additional JPA Mappings” \[508\]](#) in the Reference Guide for details.

12.6.1. Single Table

The `InheritanceType.SINGLE_TABLE` strategy maps all classes in the hierarchy to the base class' table.



In our model, `Subscription` is mapped to the `CNTRCT.SUB` table. `LifetimeSubscription`, which extends `Subscription`, adds its field data to this table as well.

Example 12.5. Single Table Mapping

```
@Entity
@Table(name="SUB", schema="CNTRCT")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Subscription
{
    ...
}

@Entity(name="Lifetime")
public class LifetimeSubscription
    extends Subscription
{

```

```
} ...
```

The same metadata expressed in XML form:

```
<entity class="org.mag.subscribe.Subscription">
  <table name="SUB" schema="CNTRCT"/>
  <inheritance strategy="SINGLE_TABLE"/>
  ...
</entity>
<entity class="org.mag.subscribe.LifetimeSubscription">
  ...
</entity>
```

Single table inheritance is the default strategy. Thus, we could omit the `@Inheritance` annotation in the example above and get the same result.

Note

Mapping subclass state to the superclass table is often called *flat* inheritance mapping.

12.6.1.1. Advantages

Single table inheritance mapping is the fastest of all inheritance models, since it never requires a join to retrieve a persistent instance from the database. Similarly, persisting or updating a persistent instance requires only a single `INSERT` or `UPDATE` statement. Finally, relations to any class within a single table inheritance hierarchy are just as efficient as relations to a base class.

12.6.1.2. Disadvantages

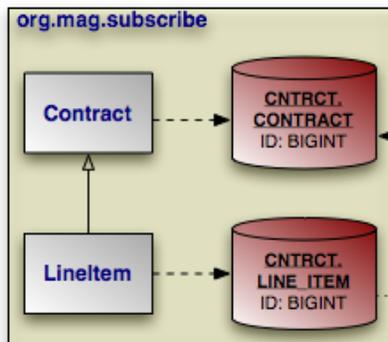
The larger the inheritance model gets, the "wider" the mapped table gets, in that for every field in the entire inheritance hierarchy, a column must exist in the mapped table. This may have undesirable consequence on the database size, since a wide or deep inheritance hierarchy will result in tables with many mostly-empty columns.

12.6.2. Joined

The `InheritanceType.JOINED` strategy uses a different table for each class in the hierarchy. Each table only includes state declared in its class. Thus to load a subclass instance, the EJB persistence implementation must read from the subclass table as well as the table of each ancestor class, up to the base entity class.

Note

Using joined subclass tables is also called *vertical* inheritance mapping.



PrimaryKeyJoinColumn annotations tell the EJB implementation how to join each subclass table record to the corresponding record in its direct superclass table. In our model, the `LINE_ITEM.ID` column joins to the `CONTRACT.ID` column. The `PrimaryKeyJoinColumn` annotation has the following properties:

- `String name`: The name of the subclass table column. When there is a single identity field, defaults to that field's column name.
- `String referencedColumnName`: The name of the superclass table column this subclass table column joins to. When there is a single identity field, defaults to that field's column name.
- `String columnDefinition`: This property has the same meaning as the `columnDefinition` property on the `Column` annotation, described in [Section 12.3, "Column" \[123\]](#).

The XML equivalent is the `primary-key-join-column` element. Its attributes mirror the annotation properties described above:

- `name`
- `referenced-column-name`
- `column-definition`

The example below shows how we use `InheritanceTable.JOINED` and a primary key join column to map our sample model according to the diagram above. Note that a primary key join column is not strictly needed, because there is only one identity column, and the subclass table column has the same name as the superclass table column. In this situation, the defaults suffice. However, we include the primary key join column for illustrative purposes.

Example 12.6. Joined Subclass Tables

```

@Entity
@Table(schema="CNRCT")
@Inheritance(strategy=InheritanceType.JOINED)
public class Contract
    extends Document
{
    ...
}

public class Subscription
{
    ...

    @Entity
    @Table(name="LINE_ITEM", schema="CNRCT")
    @PrimaryKeyJoinColumn(name="ID", referencedColumnName="ID")

```

```

public static class LineItem
    extends Contract
{
    ...
}

```

The same metadata expressed in XML form:

```

<entity class="org.mag.subscribe.Contract">
  <table schema="CNRCT"/>
  <inheritance strategy="JOINED"/>
  ...
</entity>
<entity class="org.mag.subscribe.Subscription.LineItem">
  <table name="LINE_ITEM" schema="CNRCT"/>
  <primary-key-join-column name="ID" referenced-column-name="PK"/>
  ...
</entity>

```

When there are multiple identity columns, you must define multiple `PrimaryKeyJoinColumns` using the aptly-named `PrimaryKeyJoinColumns` annotation. This annotation's value is an array of `PrimaryKeyJoinColumns`. We could rewrite `LineItem`'s mapping as:

```

@Entity
@Table(name="LINE_ITEM", schema="CNRCT")
@PrimaryKeyJoinColumns({
    @PrimaryKeyJoinColumn(name="ID", referencedColumnName="ID")
})
public static class LineItem
    extends Contract
{
    ...
}

```

In XML, simply list as many `primary-key-join-column` elements as necessary.

12.6.2.1. Advantages

The joined strategy has the following advantages:

1. Using joined subclass tables results in the most *normalized* database schema, meaning the schema with the least spurious or redundant data.
2. As more subclasses are added to the data model over time, the only schema modification that needs to be made is the addition of corresponding subclass tables in the database (rather than having to change the structure of existing tables).
3. Relations to a base class using this strategy can be loaded through standard joins and can use standard foreign keys, as opposed to the machinations required to load polymorphic relations to table-per-class base types, described below.

12.6.2.2. Disadvantages

Aside from certain uses of the table-per-class strategy described below, the joined strategy is often the slowest of the inheritance models. Retrieving any subclass requires one or more database joins, and storing subclasses requires multiple `INSERT` or `UP-`

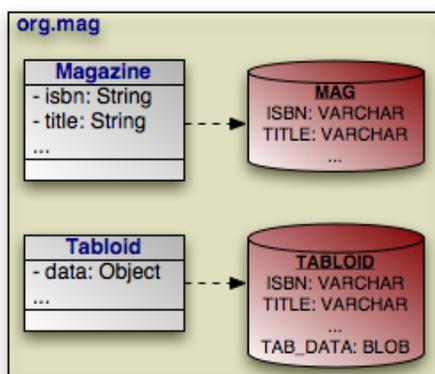
DATE statements. This is only the case when persistence operations are performed on subclasses; if most operations are performed on the least-derived persistent superclass, then this mapping is very fast.

Note

When executing a select against a hierarchy that uses joined subclass table inheritance, you must consider how to load subclass state. [Section 5.7, “Eager Fetching” \[473\]](#) in the Reference Guide describes Kodo's options for efficient data loading.

12.6.3. Table Per Class

Like the JOINED strategy, the `InheritanceType.TABLE_PER_CLASS` strategy uses a different table for each class in the hierarchy. Unlike the JOINED strategy, however, each table includes all state for an instance of the corresponding class. Thus to load a subclass instance, the EJB persistence implementation must only read from the subclass table; it does not need to join to superclass tables.



Suppose that our sample model's Magazine class has a subclass Tabloid. The classes are mapped using the table-per-class strategy, as in the diagram above. In a table-per-class mapping, Magazine's table MAG contains all state declared in the base Magazine class. Tabloid maps to a separate table, TABLOID. This table contains not only the state declared in the Tabloid subclass, but all the base class state from Magazine as well. Thus the TABLOID table would contain columns for isbn, title, and other Magazine fields. These columns would default to the names used in Magazine's mapping metadata. [Section 12.8.3, “Embedded Mapping” \[150\]](#) will show you how to use `AttributeOverrides` and `AssociationOverrides` to override superclass field mappings.

Example 12.7. Table Per Class Mapping

```

@Entity
@Table(name="MAG")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Magazine
{
    ...
}

@Entity
@Table(name="TABLOID")
public class Tabloid
    extends Magazine
{
    ...
}
  
```

And the same classes in XML:

```
<entity class="org.mag.Magazine">
  <table name="MAG"/>
  <inheritance strategy="TABLE_PER_CLASS"/>
  ...
</entity>
<entity class="org.mag.Tabloid">
  <table name="TABLOID"/>
  ...
</entity>
```

12.6.3.1. Advantages

The table-per-class strategy is very efficient when operating on instances of a known class. Under these conditions, the strategy never requires joining to superclass or subclass tables. Reads, joins, inserts, updates, and deletes are all efficient in the absence of polymorphic behavior. Also, as in the joined strategy, adding additional classes to the hierarchy does not require modifying existing class tables.

12.6.3.2. Disadvantages

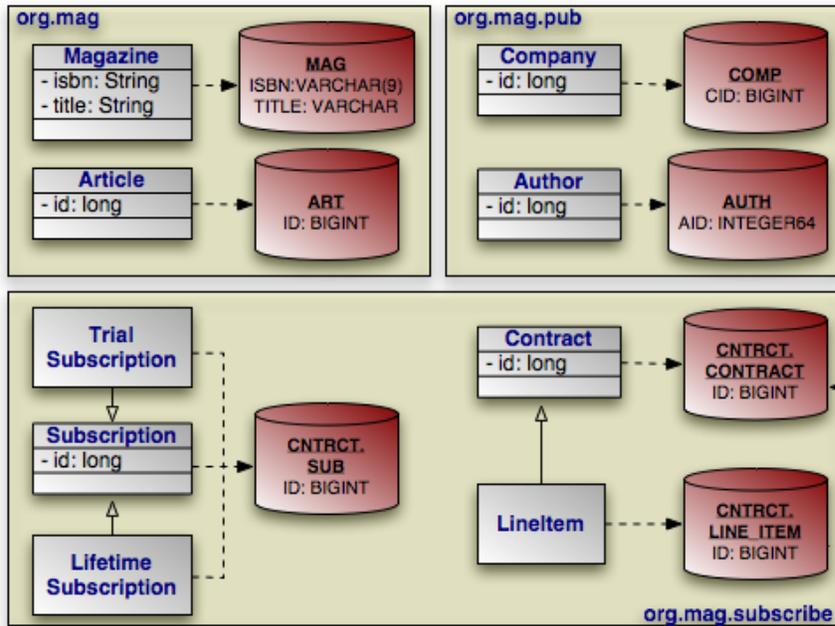
Polymorphic relations to non-leaf classes in a table-per-class hierarchy have many limitations. When the concrete subclass is not known, the related object could be in any of the subclass tables, making joins through the relation impossible. This ambiguity also affects identity lookups and queries; these operations require multiple SQL SELECTs (one for each possible subclass), or a complex UNION.

Note

Section 7.8.1, “Table Per Class” [519] in the Reference Guide describes the limitations Kodo places on table-per-class mapping.

12.6.4. Putting it All Together

Now that we have covered EJB's inheritance strategies, we can update our mapping document with inheritance information. Here is the complete model:



And here is the corresponding mapping metadata:

Example 12.8. Inheritance Mapping

```

package org.mag;

@Entity
@IdClass(Magazine.MagazineId.class)
@Table(name="MAG")
public class Magazine
{
    @Column(length=9)
    @Id private String isbn;
    @Id private String title;

    ...

    public static class MagazineId
    {
        ...
    }
}

@Entity
@Table(name="ART", uniqueConstraints=@Unique(columnNames="TITLE"))
@SequenceGenerator(name="ArticleSeq", sequenceName="ART_SEQ")
public class Article
{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="ArticleSeq")
    private long id;

    ...
}

package org.mag.pub;

@Entity
@Table(name="COMP")
public class Company
{
    @Column(name="CID")
    @Id private long id;

    ...
}

@Entity

```

```

@Table(name="AUTH")
public class Author
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="AuthorGen")
    @TableGenerator(name="AuthorGen", table="AUTH_GEN", pkColumnName="PK",
        valueColumnName="AID")
    @Column(name="AID", columnDefinition="INTEGER64")
    private long id;

    ...
}

@Embeddable
public class Address
{
    ...
}

package org.mag.subscribe;

@MappedSuperclass
public abstract class Document
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

    ...
}

@Entity
@Table(schema="CNTRCT")
@Inheritance(strategy=InheritanceType.JOINED)
public class Contract
    extends Document
{
    ...
}

@Entity
@Table(name="SUB", schema="CNTRCT")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Subscription
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

    ...

    @Entity
    @Table(name="LINE_ITEM", schema="CNTRCT")
    @PrimaryKeyJoinColumn(name="ID", referencedColumnName="ID")
    public static class LineItem
        extends Contract
    {
        ...
    }
}

@Entity(name="Lifetime")
public class LifetimeSubscription
    extends Subscription
{
    ...
}

@Entity(name="Trial")
public class TrialSubscription
    extends Subscription
{
    ...
}

```

The same metadata expressed in XML form:

```

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
    version="1.0">
    <mapped-superclass class="org.mag.subscribe.Document">
        <attributes>
            <id name="id">
                <generated-value strategy="IDENTITY"/>
            </id>
        </attributes>
    </mapped-superclass>

```

```

        </id>
        ...
    </attributes>
</mapped-superclass>
<entity class="org.mag.Magazine">
    <table name="MAG"/>
    <id-class="org.mag.Magazine.MagazineId"/>
    <attributes>
        <id name="isbn">
            <column length="9"/>
        </id>
        <id name="title"/>
        ...
    </attributes>
</entity>
<entity class="org.mag.Article">
    <table name="ART">
        <unique-constraint>
            <column-name>TITLE</column-name>
        </unique-constraint>
    </table>
    <sequence-generator name="ArticleSeq" sequence-name="ART_SEQ"/>
    <attributes>
        <id name="id">
            <generated-value strategy="SEQUENCE" generator="ArticleSeq"/>
        </id>
        ...
    </attributes>
</entity>
<entity class="org.mag.pub.Company">
    <table name="COMP"/>
    <attributes>
        <id name="id">
            <column name="CID"/>
        </id>
        ...
    </attributes>
</entity>
<entity class="org.mag.pub.Author">
    <table name="AUTH"/>
    <attributes>
        <id name="id">
            <column name="AID" column-definition="INTEGER64"/>
            <generated-value strategy="TABLE" generator="AuthorGen"/>
            <table-generator name="AuthorGen" table="AUTH_GEN"
                pk-column-name="PK" value-column-name="AID"/>
        </id>
        ...
    </attributes>
</entity>
<entity class="org.mag.subscribe.Contract">
    <table schema="CNTRCT"/>
    <inheritance strategy="JOINED"/>
    <attributes>
        ...
    </attributes>
</entity>
<entity class="org.mag.subscribe.Subscription">
    <table name="SUB" schema="CNTRCT"/>
    <inheritance strategy="SINGLE_TABLE"/>
    <attributes>
        <id name="id">
            <generated-value strategy="IDENTITY"/>
        </id>
        ...
    </attributes>
</entity>
<entity class="org.mag.subscribe.Subscription.LineItem">
    <table name="LINE_ITEM" schema="CNTRCT"/>
    <primary-key-join-column name="ID" referenced-column-name="PK"/>
    ...
</entity>
<entity class="org.mag.subscribe.LifetimeSubscription" name="Lifetime">
    ...
</entity>
<entity class="org.mag.subscribe.TrialSubscription" name="Trial">
    ...
</entity>
</entity-mappings>

```

12.7. Discriminator

The **single table** inheritance strategy results in a single table containing records for two or more different classes in an inheritance hierarchy. Similarly, using the **joined** strategy results in the superclass table holding records for superclass instances as well as for the superclass state of subclass instances. When selecting data, EJB needs a way to differentiate a row representing an object of one class from a row representing an object of another. That is the job of the *discriminator* column.

The discriminator column is always in the table of the base entity. It holds a different value for records of each class, allowing the EJB persistence runtime to determine what class of object each row represents.

The `DiscriminatorColumn` annotation represents a discriminator column. It has these properties:

- `String name`: The column name. Defaults to `DTYPE`.
- `length`: For string discriminator values, the length of the column. Defaults to 31.
- `String columnDefinition`: This property has the same meaning as the `columnDefinition` property on the `Column` annotation, described in [Section 12.3, “Column” \[123\]](#).
- `DiscriminatorType discriminatorType`: Enum value declaring the discriminator strategy of the hierarchy.

The corresponding XML element is `discriminator-column`. Its attributes mirror the annotation properties above:

- `name`
- `length`
- `column-definition`
- `discriminator-type`: One of `STRING`, `CHAR`, or `INTEGER`.

The `DiscriminatorValue` annotation specifies the discriminator value for each class. Though this annotation's value is always a string, the implementation will parse it according to the `DiscriminatorColumn`'s `discriminatorType` property above. The type defaults to `DiscriminatorType.STRING`, but may be `DiscriminatorType.CHAR` or `DiscriminatorType.INTEGER`. If you do not specify a `DiscriminatorValue`, the provider will choose an appropriate default.

The corresponding XML element is `discriminator-value`. The text within this element is parsed as the discriminator value.

Note

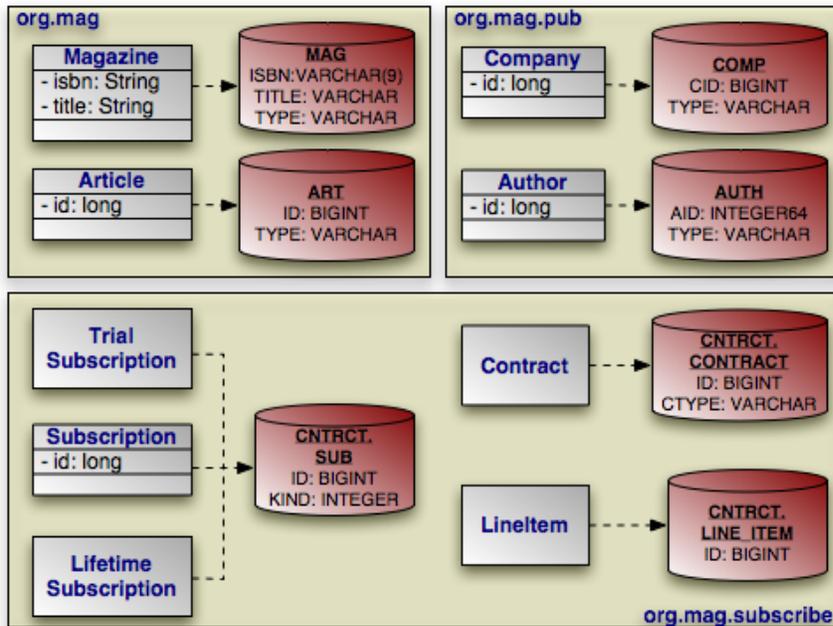
Kodo assumes your model employs a discriminator column if any of the following are true:

1. The base entity explicitly declares an inheritance type of `SINGLE_TABLE`.
2. The base entity sets a discriminator value.
3. The base entity declares a discriminator column.

Only `SINGLE_TABLE` inheritance hierarchies require a discriminator column and values. `JOINED` hierarchies can use a discriminator to make some operations more efficient, but do not require one. `TABLE_PER_CLASS` hierarchies have no use for a discriminator.

Kodo defines additional discriminator strategies; see [Section 7.7, “Additional JPA Mappings” \[508\]](#) in the Reference Guide for details. Kodo also supports final entity classes. Kodo does not use a discriminator on final classes.

We can now translate our newfound knowledge of EJB discriminators into concrete EJB mappings. We first extend our diagram with discriminator columns:



Next, we present the updated mapping document. Notice that in this version, we have removed explicit inheritance annotations when the defaults sufficed. Also, notice that entities using the default DTYPE discriminator column mapping do not need an explicit DiscriminatorColumn annotation.

Example 12.9. Discriminator Mapping

```

package org.mag;

@Entity
@IdClass(Magazine.MagazineId.class)
@Table(name="MAG")
@DiscriminatorValue("Mag")
public class Magazine
{
    @Column(length=9)
    @Id private String isbn;
    @Id private String title;

    ...

    public static class MagazineId
    {
        ...
    }
}

@Entity
@Table(name="ART", uniqueConstraints=@Unique(columnNames="TITLE"))
@SequenceGenerator(name="ArticleSeq", sequenceName="ART_SEQ")
public class Article
{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="ArticleSeq")
    private long id;

    ...
}

package org.mag.pub;

@Entity
@Table(name="COMP")

```

```

public class Company
{
    @Column(name="CID")
    @Id private long id;

    ...
}

@Entity
@Table(name="AUTH")
public class Author
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="AuthorGen")
    @TableGenerator(name="AuthorGen", table="AUTH_GEN", pkColumnName="PK",
        valueColumnName="AID")
    @Column(name="AID", columnDefinition="INTEGER64")
    private long id;

    ...
}

@Embeddable
public class Address
{
    ...
}

package org.mag.subscribe;

@MappedSuperclass
public abstract class Document
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

    ...
}

@Entity
@Table(schema="CNTRCT")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="CTYPE")
public class Contract
    extends Document
{
    ...
}

@Entity
@Table(name="SUB", schema="CNTRCT")
@DiscriminatorColumn(name="KIND", discriminatorType=DiscriminatorType.INTEGER)
@DiscriminatorValue("1")
public class Subscription
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

    ...

    @Entity
    @Table(name="LINE_ITEM", schema="CNTRCT")
    public static class LineItem
        extends Contract
    {
        ...
    }
}

@Entity(name="Lifetime")
@DiscriminatorValue("2")
public class LifetimeSubscription
    extends Subscription
{
    ...
}

@Entity(name="Trial")
@DiscriminatorValue("3")
public class TrialSubscription
    extends Subscription
{
    ...
}

```

The same metadata expressed in XML:

```

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0">
  <mapped-superclass class="org.mag.subscribe.Document">
    <attributes>
      <id name="id">
        <generated-value strategy="IDENTITY"/>
      </id>
      ...
    </attributes>
  </mapped-superclass>
  <entity class="org.mag.Magazine">
    <table name="MAG"/>
    <id-class="org.mag.Magazine.MagazineId"/>
    <discriminator-value>Mag</discriminator-value>
    <attributes>
      <id name="isbn">
        <column length="9"/>
      </id>
      <id name="title"/>
      ...
    </attributes>
  </entity>
  <entity class="org.mag.Article">
    <table name="ART">
      <unique-constraint>
        <column-name>TITLE</column-name>
      </unique-constraint>
    </table>
    <sequence-generator name="ArticleSeq" sequence-name="ART_SEQ"/>
    <attributes>
      <id name="id">
        <generated-value strategy="SEQUENCE" generator="ArticleSeq"/>
      </id>
      ...
    </attributes>
  </entity>
  <entity class="org.mag.pub.Company">
    <table name="COMP"/>
    <attributes>
      <id name="id">
        <column name="CID"/>
      </id>
      ...
    </attributes>
  </entity>
  <entity class="org.mag.pub.Author">
    <table name="AUTH"/>
    <attributes>
      <id name="id">
        <column name="AID" column-definition="INTEGER64"/>
        <generated-value strategy="TABLE" generator="AuthorGen"/>
        <table-generator name="AuthorGen" table="AUTH_GEN"
          pk-column-name="PK" value-column-name="AID"/>
      </id>
      ...
    </attributes>
  </entity>
  <entity class="org.mag.subscribe.Contract">
    <table schema="CNTRCT"/>
    <inheritance strategy="JOINED"/>
    <discriminator-column name="CTYPE"/>
    <attributes>
      ...
    </attributes>
  </entity>
  <entity class="org.mag.subscribe.Subscription">
    <table name="SUB" schema="CNTRCT"/>
    <inheritance strategy="SINGLE_TABLE"/>
    <discriminator-value>1</discriminator-value>
    <discriminator-column name="KIND" discriminator-type="INTEGER"/>
    <attributes>
      <id name="id">
        <generated-value strategy="IDENTITY"/>
      </id>
      ...
    </attributes>
  </entity>
  <entity class="org.mag.subscribe.Subscription.LineItem">
    <table name="LINE_ITEM" schema="CNTRCT"/>
    <primary-key-join-column name="ID" referenced-column-name="PK"/>
    ...
  </entity>
  <entity class="org.mag.subscribe.LifetimeSubscription" name="Lifetime">
    <discriminator-value>2</discriminator-value>
    ...
  </entity>
  <entity class="org.mag.subscribe.TrialSubscription" name="Trial">

```

```

    <discriminator-value>3</discriminator-value>
    ...
  </entity>
</entity-mappings>

```

12.8. Field Mapping

The following sections enumerate the myriad of field mappings EJB persistence supports. EJB augments the persistence metadata covered in [Chapter 5, *Metadata* \[28\]](#) with many new object-relational annotations. As we explore the library of standard mappings, we introduce each of these enhancements in context.

Note

Kodo supports many additional field types, and allows you to create custom mappings for unsupported field types or database schemas. See the Reference Guide's [Chapter 7, *Mapping* \[489\]](#) for complete coverage of Kodo EJB's mapping capabilities.

12.8.1. Basic Mapping

A *basic* field mapping stores the field value directly into a database column. The following field metadata types use basic mapping. These types were defined in [Section 5.2, “Field and Property Metadata” \[33\]](#).

- **Id** fields.
- **Version** fields.
- **Basic** fields.

In fact, you have already seen examples of basic field mappings in this chapter - the mapping of all identity fields in [Example 12.3, “Identity Mapping” \[124\]](#). As you saw in that section, to write a basic field mapping you use the `Column` annotation to describe the column the field value is stored in. We discussed the `Column` annotation in [Section 12.3, “Column” \[123\]](#). Recall that the name of the column defaults to the field name, and the type of the column defaults to an appropriate type for the field type. These defaults allow you to sometimes omit the annotation altogether.

12.8.1.1. LOBs

Adding the `Lob` marker annotation to a basic field signals that the data is to be stored as a LOB (Large Object). If the field holds string or character data, it will map to a CLOB (Character Large Object) database column. If the field holds any other data type, it will be stored as binary data in a BLOB (Binary Large Object) column. The implementation will serialize the Java value if needed.

The equivalent XML element is `lob`, which has no children or attributes.

12.8.1.2. Enumerated

You can apply the `Enumerated` annotation to your `Enum` fields to control how they map to the database. The `Enumerated` annotation's value one of the following constants from the `EnumType` enum:

- `EnumType.ORDINAL`: The default. The persistence implementation places the ordinal value of the enum in a numeric column. This is an efficient mapping, but may break if you rearrange the Java enum declaration.

- `EnumType.STRING`: Store the name of the enum value rather than the ordinal. This mapping uses a `VARCHAR` column rather than a numeric one.

The `Enumerated` annotation is optional. Any un-annotated enumeration field defaults to `ORDINAL` mapping.

The corresponding XML element is `enumerated`. Its embedded text must be one of `STRING` or `ORIDINAL`.

12.8.1.3. Temporal Types

The `Temporal` annotation determines how the implementation handles your basic `java.util.Date` and `java.util.Calendar` fields at the JDBC level. The `Temporal` annotation's value is a constant from the `TemporalType` enum. Available values are:

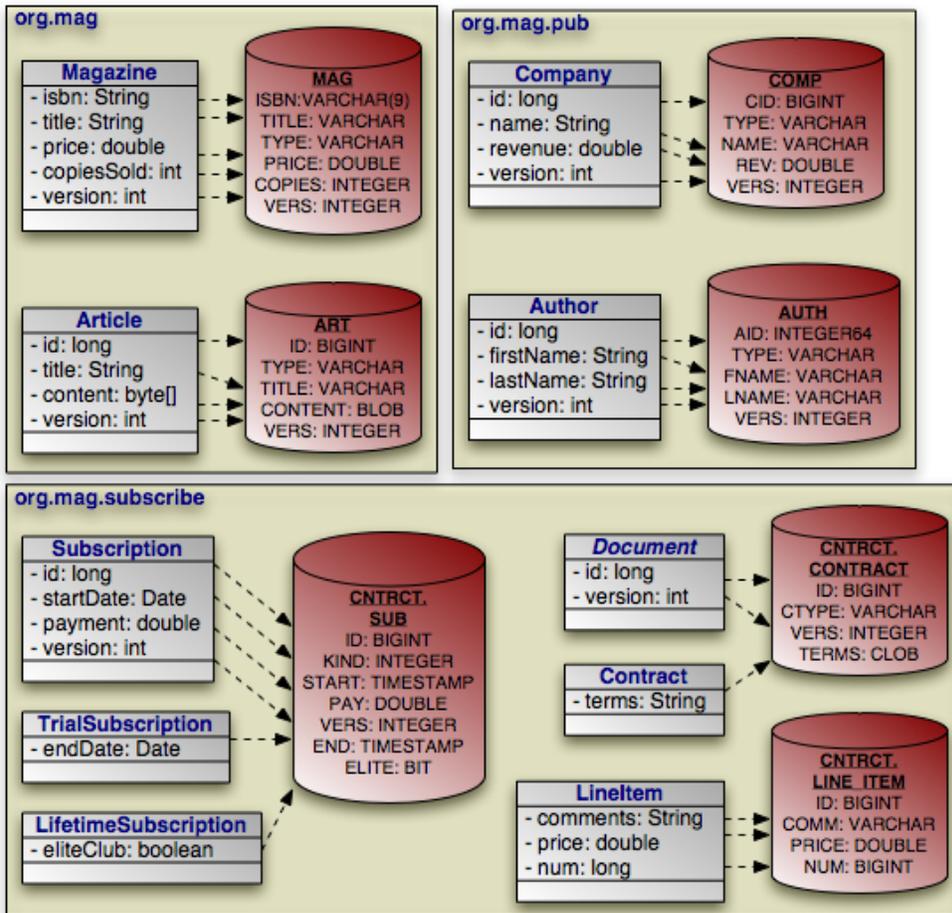
- `TemporalType.TIMESTAMP`: The default. Use JDBC's timestamp APIs to manipulate the column data.
- `TemporalType.DATE`: Use JDBC's SQL date APIs to manipulate the column data.
- `TemporalType.TIME`: Use JDBC's time APIs to manipulate the column data.

If the `Temporal` annotation is omitted, the implementation will treat the data as a timestamp.

The corresponding XML element is `temporal`, whose text value must be one of: `TIME`, `DATE`, or `TIMESTAMP`.

12.8.1.4. The Updated Mappings

Below we present an updated diagram of our model and its associated database schema, followed by the corresponding mapping metadata. Note that the mapping metadata relies on defaults where possible. Also note that as a mapped superclass, `Document` can define mappings that will automatically transfer to its subclass' tables. In [Section 12.8.3, “Embedded Mapping” \[150\]](#) you will see how a subclass can override its mapped superclass' mappings.



Example 12.10. Basic Field Mapping

```

package org.mag;

@Entity
@IdClass(Magazine.MagazineId.class)
@Table(name="MAG")
@DiscriminatorValue("Mag")
public class Magazine
{
    @Column(length=9)
    @Id private String isbn;
    @Id private String title;

    @Column(name="VERS")
    @Version private int version;

    private String name;
    private double price;

    @Column(name="COPIES")
    private int copiesSold;

    ...

    public static class MagazineId
    {
        ...
    }
}

@Entity
@Table(name="ART", uniqueConstraints=@Unique(columnNames="TITLE"))
@SequenceGenerator(name="ArticleSeq", sequenceName="ART_SEQ")

```

```

public class Article
{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="ArticleSeq")
    private long id;

    @Column(name="VERS")
    @Version private int version;

    private String title;
    private byte[] content;

    ...
}

package org.mag.pub;

@Entity
@Table(name="COMP")
public class Company
{
    @Column(name="CID")
    @Id private long id;

    @Column(name="VERS")
    @Version private int version;

    private String name;

    @Column(name="REV")
    private double revenue;

    ...
}

@Entity
@Table(name="AUTH")
public class Author
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="AuthorGen")
    @TableGenerator(name="AuthorGen", table="AUTH_GEN", pkColumnName="PK",
        valueColumnName="AID")
    @Column(name="AID", columnDefinition="INTEGER64")
    private long id;

    @Column(name="VERS")
    @Version private int version;

    @Column(name="FNAME")
    private String firstName;

    @Column(name="LNAME")
    private String lastName;

    ...
}

@Embeddable
public class Address
{
    ...
}

package org.mag.subscribe;

@MappedSuperclass
public abstract class Document
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

    @Column(name="VERS")
    @Version private int version;

    ...
}

@Entity
@Table(schema="CNTRCT")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="CTYPE")
public class Contract
    extends Document
{
    @Lob
    private String terms;

    ...
}

```

```

@Entity
@Table(name="SUB", schema="CNTRCT")
@DiscriminatorColumn(name="KIND", discriminatorType=DiscriminatorType.INTEGER)
@DiscriminatorValue("1")
public class Subscription
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

    @Column(name="VERS")
    @Version private int version;

    @Column(name="START")
    private Date startDate;

    @Column(name="PAY")
    private double payment;

    ...

    @Entity
    @Table(name="LINE_ITEM", schema="CNTRCT")
    public static class LineItem
        extends Contract
    {
        @Column(name="COMM")
        private String comments;

        private double price;
        private long num;
        ...
    }
}

@Entity(name="Lifetime")
@DiscriminatorValue("2")
public class LifetimeSubscription
    extends Subscription
{
    @Basic(fetch=FetchType.LAZY)
    @Column(name="ELITE")
    private boolean getEliteClub () { ... }
    public void setEliteClub (boolean elite) { ... }

    ...
}

@Entity(name="Trial")
@DiscriminatorValue("3")
public class TrialSubscription
    extends Subscription
{
    @Column(name="END")
    public Date getEndDate () { ... }
    public void setEndDate (Date end) { ... }

    ...
}

```

The same metadata expressed in XML:

```

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
version="1.0">
<mapped-superclass class="org.mag.subscribe.Document">
    <attributes>
        <id name="id">
            <generated-value strategy="IDENTITY"/>
        </id>
        <version name="version">
            <column name="VERS"/>
        </version>
        ...
    </attributes>
</mapped-superclass>
<entity class="org.mag.Magazine">
    <table name="MAG"/>
    <id-class="org.mag.Magazine.MagazineId"/>
    <discriminator-value>Mag</discriminator-value>
    <attributes>
        <id name="isbn">
            <column length="9"/>
        </id>
        <id name="title"/>
    </attributes>

```

```

        <basic name="name"/>
        <basic name="price"/>
        <basic name="copiesSold">
            <column name="COPIES"/>
        </basic>
        <version name="version">
            <column name="VERS"/>
        </version>
        ...
    </attributes>
</entity>
<entity class="org.mag.Article">
    <table name="ART">
        <unique-constraint>
            <column-name>TITLE</column-name>
        </unique-constraint>
    </table>
    <sequence-generator name="ArticleSeq", sequenceName="ART_SEQ"/>
    <attributes>
        <id name="id">
            <generated-value strategy="SEQUENCE" generator="ArticleSeq"/>
        </id>
        <basic name="title"/>
        <basic name="content"/>
        <version name="version">
            <column name="VERS"/>
        </version>
        ...
    </attributes>
</entity>
<entity class="org.mag.pub.Company">
    <table name="COMP"/>
    <attributes>
        <id name="id">
            <column name="CID"/>
        </id>
        <basic name="name"/>
        <basic name="revenue">
            <column name="REV"/>
        </basic>
    </attributes>
</entity>
<entity class="org.mag.pub.Author">
    <table name="AUTH"/>
    <attributes>
        <id name="id">
            <column name="AID" column-definition="INTEGER64"/>
            <generated-value strategy="TABLE" generator="AuthorGen"/>
            <table-generator name="AuthorGen" table="AUTH_GEN"
                pk-column-name="PK" value-column-name="AID"/>
        </id>
        <basic name="firstName">
            <column name="FNAME"/>
        </basic>
        <basic name="lastName">
            <column name="LNAME"/>
        </basic>
        <version name="version">
            <column name="VERS"/>
        </version>
        ...
    </attributes>
</entity>
<entity class="org.mag.subscribe.Contract">
    <table schema="CNTRCT"/>
    <inheritance strategy="JOINED"/>
    <discriminator-column name="CTYPE"/>
    <attributes>
        <basic name="terms">
            <lob/>
        </basic>
        ...
    </attributes>
</entity>
<entity class="org.mag.subscribe.Subscription">
    <table name="SUB" schema="CNTRCT"/>
    <inheritance strategy="SINGLE_TABLE"/>
    <discriminator-value>1</discriminator-value>
    <discriminator-column name="KIND" discriminator-type="INTEGER"/>
    <attributes>
        <id name="id">
            <generated-value strategy="IDENTITY"/>
        </id>
        <basic name="payment">
            <column name="PAY"/>
        </basic>
        <basic name="startDate">
            <column name="START"/>
        </basic>
        <version name="version">
            <column name="VERS"/>
        </version>
        ...
    </attributes>

```

```

    </attributes>
  </entity>
  <entity class="org.mag.subscribe.Subscription.LineItem">
    <table name="LINE_ITEM" schema="CNTRCT"/>
    <primary-key-join-column name="ID" referenced-column-name="PK"/>
    <attributes>
      <basic name="comments">
        <column name="COMM"/>
      </basic>
      <basic name="price"/>
      <basic name="num"/>
      ...
    </attributes>
  </entity>
  <entity class="org.mag.subscribe.LifetimeSubscription" name="Lifetime">
    <discriminator-value>2</discriminator-value>
    <attributes>
      <basic name="eliteClub" fetch="LAZY">
        <column name="ELITE"/>
      </basic>
      ...
    </attributes>
  </entity>
  <entity class="org.mag.subscribe.TrialSubscription" name="Trial">
    <discriminator-value>3</discriminator-value>
    <attributes>
      <basic name="endDate">
        <column name="END"/>
      </basic>
      ...
    </attributes>
  </entity>
</entity-mappings>

```

12.8.2. Secondary Tables

Sometimes a logical record is spread over multiple database tables. EJB persistence calls a class' declared table the *primary* table, and calls other tables that make up a logical record *secondary* tables. You can map any persistent field to a secondary table. Just write the standard field mapping, then perform these two additional steps:

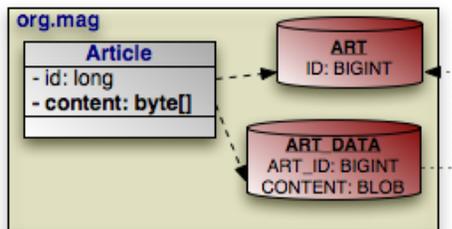
1. Set the `table` attribute of each of the field's columns or join columns to the name of the secondary table.
2. Define the secondary table on the entity class declaration.

You define secondary tables with the `SecondaryTable` annotation. This annotation has all the properties of the `Table` annotation covered in [Section 12.1, “Table” \[119\]](#), plus a `pkJoinColumns` property.

The `pkJoinColumns` property is an array of `PrimaryKeyJoinColumn` dictating how to join secondary table records to their owning primary table records. Each `PrimaryKeyJoinColumn` joins a secondary table column to a primary key column in the primary table. See [Section 12.6.2, “Joined” \[131\]](#) above for coverage of `PrimaryKeyJoinColumn`'s properties.

The corresponding XML element is `secondary-table`. This element has all the attributes of the `table` element, but also accepts nested `primary-key-join-column` elements.

In the following example, we move the `Article.content` field we mapped in [Section 12.8.1, “Basic Mapping” \[143\]](#) into a joined secondary table, like so:



Example 12.11. Secondary Table Field Mapping

```
package org.mag;

@Entity
@Table(name="ART")
@SecondaryTable(name="ART_DATA",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="ART_ID", referencedColumnName="ID"))
public class Article
{
    @Id private long id;

    @Column(table="ART_DATA")
    private byte[] content;

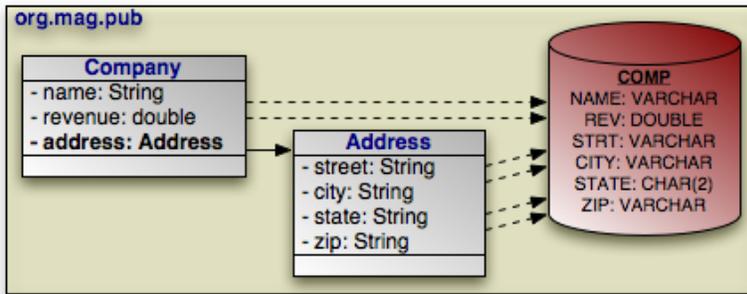
    ...
}
```

And in XML:

```
<entity class="org.mag.Article">
  <table name="ART">
    <secondary-table name="ART_DATA">
      <primary-key-join-column name="ART_ID" referenced-column-name="ID" />
    </secondary-table>
  <attributes>
    <id name="id"/>
    <basic name="content">
      <column table="ART_DATA"/>
    </basic>
    ...
  </attributes>
</entity>
```

12.8.3. Embedded Mapping

Chapter 5, *Metadata* [28] describes EJB's concept of *embeddable* objects. The field values of embedded objects are stored as part of the owning record, rather than as a separate database record. Thus, instead of mapping a relation to an embeddable object as a foreign key, you map all the fields of the embeddable instance to columns in the owning field's table.



EJB persistence defaults the embedded column names and descriptions to those of the embeddable class' field mappings. The `AttributeOverride` annotation overrides a basic embedded mapping. This annotation has the following properties:

- `String name`: The name of the embedded class' field being mapped to this class' table.
- `Column column`: The column defining the mapping of the embedded class' field to this class' table.

The corresponding XML element is `attribute-override`. It has a single `name` attribute to name the field being overridden, and a single `column` child element.

To declare multiple overrides, use the `AttributeOverrides` annotation, whose value is an array of `AttributeOverrides`. In XML, simply list multiple `attribute-override` elements in succession.

To override a many to one or one to one relationship, use the `AssociationOverride` annotation in place of `AttributeOverride`. `AssociationOverride` has the following properties:

- `String name`: The name of the embedded class' field being mapped to this class' table.
- `JoinColumn[] joinColumns`: The foreign key columns joining to the related record.

The corresponding XML element is `association-override`. It has a single `name` attribute to name the field being overridden, and one or more `join-column` child elements.

To declare multiple relation overrides, use the `AssociationOverrides` annotation, whose value is an array of `AssociationOverrides`. In XML, simply list multiple `association-override` elements in succession.

Example 12.12. Embedded Field Mapping

In this example, `Company` overrides the default mapping of `Address.street` and `Address.city`. All other embedded mappings are taken from the `Address` embeddable class.

```
package org.mag.pub;

@Entity
@Table(name="COMP")
public class Company
{
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="street", column=@Column(name="STRT")),
        @AttributeOverride(name="city", column=@Column(name="ACITY"))
    })
    private Address address;

    ...
}
```

```

@Entity
@Table(name="AUTH")
public class Author
{
    // use all defaults from Address class mappings
    private Address address;

    ...
}

@Embeddable
public class Address
{
    private String street;
    private String city;
    @Column(columnDefinition="CHAR(2)")
    private String state;
    private String zip;
}

```

The same metadata expressed in XML:

```

<entity class="org.mag.pub.Company">
  <table name="COMP"/>
  <attributes>
    ...
    <embedded name="address">
      <attribute-override name="street">
        <column name="STRT"/>
      </attribute-override>
      <attribute-override name="city">
        <column name="ACITY"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
<entity class="org.mag.pub.Author">
  <table name="AUTH"/>
  <attributes>
    <embedded name="address">
      <!-- use all defaults from Address -->
    </embedded>
  </attributes>
</entity>
<embeddable class="org.mag.pub.Address">
  <attributes>
    <basic name="street"/>
    <basic name="city"/>
    <basic name="state">
      <column column-definition="CHAR(2)"/>
    </basic>
    <basic name="zip"/>
  </attributes>
</embeddable>

```

You can also use attribute overrides on an entity class to override mappings defined by its mapped superclass or table-per-class superclass. The example below re-maps the `Document.version` field to the `Contract` table's `CVERSION` column.

Example 12.13. Mapping Mapped Superclass Field

```

@MappedSuperclass
public abstract class Document
{
    @Column(name="VERS")
    @Version private int version;

    ...
}

@Entity
@Table(schema="CNTRCT")
@Inheritance(strategy=InheritanceType.JOINED)

```

```

@DiscriminatorColumn(name="CTYPE")
@AttributeOverride(name="version", column=@Column(name="CVERSION"))
public class Contract
    extends Document
{
    ...
}
    
```

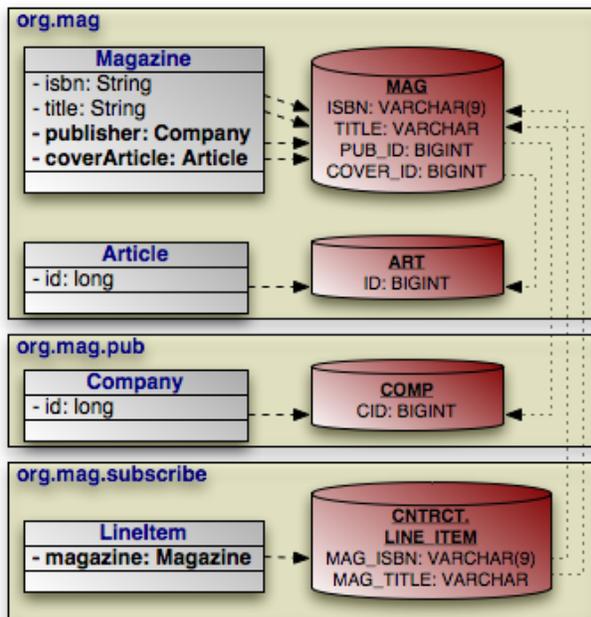
The same metadata expressed in XML form:

```

<mapped-superclass class="org.mag.subscribe.Document">
  <attributes>
    <version name="version">
      <column name="VERS">
    </version>
    ...
  </attributes>
</mapped-superclass>
<entity class="org.mag.subscribe.Contract">
  <table schema="CNTRCT"/>
  <inheritance strategy="JOINED"/>
  <discriminator-column name="CTYPE"/>
  <attribute-override name="version">
    <column name="CVERSION"/>
  </attribute-override>
  <attributes>
    ...
  </attributes>
</entity>
    
```

12.8.4. Direct Relations

A direct relation is a non-embedded persistent field that holds a reference to another entity. **many to one** and **one to one** metadata field types are mapped as direct relations. Our model has three direct relations: Magazine's publisher field is a direct relation to a Company, Magazine's coverArticle field is a direct relation to Article, and the LineItem.magazine field is a direct relation to a Magazine. Direct relations are represented in the database by foreign key columns:



You typically map a direct relation with `JoinColumn` annotations describing how the local foreign key columns join to the primary key columns of the related record. The `JoinColumn` annotation exposes the following properties:

- `String name`: The name of the foreign key column. Defaults to the relation field name, plus an underscore, plus the name of the referenced primary key column.
- `String referencedColumnName`: The name of the primary key column being joined to. If there is only one identity field in the related entity class, the join column name defaults to the name of the identity field's column.
- `boolean unique`: Whether this column is guaranteed to hold unique values for all rows. Defaults to false.

`JoinColumn` also has the same `nullable`, `insertable`, `updatable`, `columnDefinition`, and `table` properties as the `Column` annotation. See [Section 12.3, “Column” \[123\]](#) for details on these properties.

The `join-column` element represents a join column in XML. Its attributes mirror the above annotation's properties:

- `name`
- `referenced-column-name`
- `unique`
- `nullable`
- `insertable`
- `updatable`
- `column-definition`
- `table`

When there are multiple columns involved in the join, as when a `LineItem` references a `Magazine` in our model, the `JoinColumns` annotation allows you to specify an array of `JoinColumn` values. In XML, simply list multiple `join-column` elements.

Note

Kodo supports many non-standard joins. See [Section 7.6, “Non-Standard Joins” \[506\]](#) in the Reference Guide for details.

Example 12.14. Direct Relation Field Mapping

```
package org.mag;

@Table(name="AUTH")
public class Magazine
{
    @Column(length=9)
    @Id private String isbn;
    @Id private String title;

    @OneToOne
    @JoinColumn(name="COVER_ID" referencedColumnName="ID")
    private Article coverArticle;

    @ManyToOne
    @JoinColumn(name="PUB_ID" referencedColumnName="CID")
```

```

    private Company publisher;
    ...
}
@Table(name="ART")
public class Article
{
    @Id private long id;
    ...
}

package org.mag.pub;
@Table(name="COMP")
public class Company
{
    @Column(name="CID")
    @Id private long id;
    ...
}

package org.mag.subscribe;
public class Subscription
{
    ...
    @Table(name="LINE_ITEM", schema="CNTRCT")
    public static class LineItem
        extends Contract
    {
        @ManyToOne
        @JoinColumns({
            @JoinColumn(name="MAG_ISBN" referencedColumnName="ISBN"),
            @JoinColumn(name="MAG_TITLE" referencedColumnName="TITLE")
        })
        private Magazine magazine;
        ...
    }
}

```

The same metadata expressed in XML form:

```

<entity class="org.mag.Magazine">
  <table name="MAG"/>
  <id-class="org.mag.Magazine.MagazineId"/>
  <attributes>
    <id name="isbn">
      <column length="9"/>
    </id>
    <id name="title"/>
    <one-to-one name="coverArticle">
      <join-column name="COVER_ID" referenced-column-name="ID"/>
    </one-to-one>
    <many-to-one name="publisher">
      <join-column name="PUB_IC" referenced-column-name="CID"/>
    </many-to-one>
    ...
  </attributes>
</entity>
<entity class="org.mag.Article">
  <table name="ART"/>
  <attributes>
    <id name="id"/>
    ...
  </attributes>
</entity>
<entity class="org.mag.pub.Company">
  <table name="COMP"/>
  <attributes>
    <id name="id">
      <column name="CID"/>
    </id>
    ...
  </attributes>
</entity>
<entity class="org.mag.subscribe.Subscription.LineItem">
  <table name="LINE_ITEM" schema="CNTRCT"/>
  <primary-key-join-column name="ID" referenced-column-name="PK"/>
  <attributes>

```

```

<many-to-one name="magazine">
  <join-column name="MAG_ISBN" referenced-column-name="ISBN"/>
  <join-column name="MAG_TITLE" referenced-column-name="TITLE"/>
</many-to-one>
...
</attributes>
</entity>

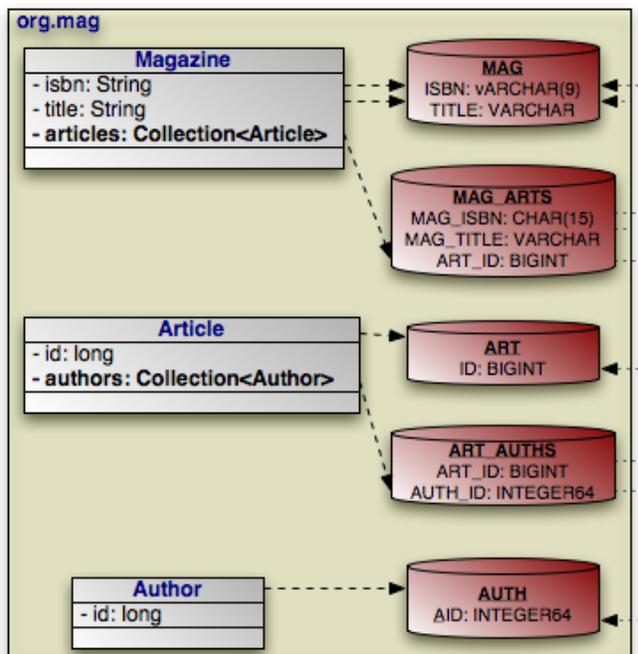
```

When the entities in a one to one relation join on shared primary key values rather than separate foreign key columns, use the `PrimaryKeyJoinColumn(s)` annotation or `primary-key-join-column` elements in place of `JoinColumn(s)` / `join-column` elements.

12.8.5. Join Table

A *join table* consists of two foreign keys. Each row of a join table associates two objects together. EJB persistence uses join tables to represent collections of entity objects: one foreign key refers back to the collection's owner, and the other refers to a collection element.

one to many and **many to many** metadata field types can map to join tables. Several fields in our model use join table mappings, including `Magazine.articles` and `Article.authors`.



You define join tables with the `JoinTable` annotation. This annotation has the following properties:

- `String name`: Table name. If not given, the name of the table defaults to the name of the owning entity's table, plus an underscore, plus the name of the related entity's table.
- `String catalog`: Table catalog.
- `String schema`: Table schema.
- `JoinColumn[] joinColumns`: Array of `JoinColumn` showing how to associate join table records with the owning

row in the primary table. This property mirrors the `pkJoinColumns` property of the `SecondaryTable` annotation in functionality. See [Section 12.8.2, “Secondary Tables” \[149\]](#) to refresh your memory on secondary tables.

If this is a bidirectional relation (see [Section 5.2.9.1, “Bidirectional Relations” \[39\]](#)), the name of a join column defaults to the inverse field name, plus an underscore, plus the referenced primary key column name. Otherwise, the join column name defaults to the field's owning entity name, plus an underscore, plus the referenced primary key column name.

- `JoinColumn[] inverseJoinColumns`: Array of `JoinColumns` showing how to associate join table records with the records that form the elements of the collection. These join columns are used just like the join columns for direct relations, and they have the same naming defaults. Read [Section 12.8.4, “Direct Relations” \[153\]](#) for a review of direct relation mapping.

`join-table` is the corresponding XML element. It has the same attributes as the `table` element, but includes the ability to nest `join-column` and `inverse-join-column` elements as children. We have seen `join-column` elements already; `inverse-join-column` elements have the same attributes.

Here are the join table mappings for the diagram above.

Example 12.15. Join Table Mapping

```
package org.mag;

@Entity
@Table(name="MAG")
public class Magazine
{
    @Column(length=9)
    @Id private String isbn;
    @Id private String title;

    @OneToMany(...)
    @OrderBy
    @JoinTable(name="MAG_ARTS",
        joinColumns={
            @JoinColumn(name="MAG_ISBN", referencedColumnName="ISBN"),
            @JoinColumn(name="MAG_TITLE", referencedColumnName="TITLE")
        },
        inverseJoinColumns=@JoinColumn(name="ART_ID", referencedColumnName="ID"))
    private Collection<Article> articles;

    ...
}

@Entity
@Table(name="ART")
public class Article
{
    @Id private long id;

    @ManyToMany(cascade=CascadeType.PERSIST)
    @OrderBy("lastName, firstName")
    @JoinTable(name="ART_AUTHS",
        joinColumns=@JoinColumn(name="ART_ID", referencedColumnName="ID"),
        inverseJoinColumns=@JoinColumn(name="AUTH_ID", referencedColumnName="AID"))
    private Collection<Author> authors;

    ...
}

package org.mag.pub;

@Entity
@Table(name="AUTH")
public class Author
{
    @Column(name="AID", columnDefinition="INTEGER64")
    @Id private long id;

    ...
}
```

The same metadata expressed in XML:

```

<entity class="org.mag.Magazine">
  <table name="MAG"/>
  <attributes>
    <id name="isbn">
      <column length="9"/>
    </id>
    <id name="title"/>
    <one-to-many name="articles">
      <order-by/>
      <join-table name="MAG_ARTS">
        <join-column name="MAG_ISBN" referenced-column-name="ISBN"/>
        <join-column name="MAG_TITLE" referenced-column-name="TITLE"/>
      </join-table>
    </one-to-many>
    ...
  </attributes>
</entity>
<entity class="org.mag.Article">
  <table name="ART"/>
  <attributes>
    <id name="id"/>
    <many-to-many name="articles">
      <order-by>lastName, firstName</order-by>
      <join-table name="ART_AUTHS">
        <join-column name="ART_ID" referenced-column-name="ID"/>
        <inverse-join-column name="AUTH_ID" referenced-column-name="AID"/>
      </join-table>
    </many-to-many>
    ...
  </attributes>
</entity>
<entity class="org.mag.pub.Author">
  <table name="AUTH"/>
  <attributes>
    <id name="id">
      <column name="AID" column-definition="INTEGER64"/>
    </id>
    ...
  </attributes>
</entity>

```

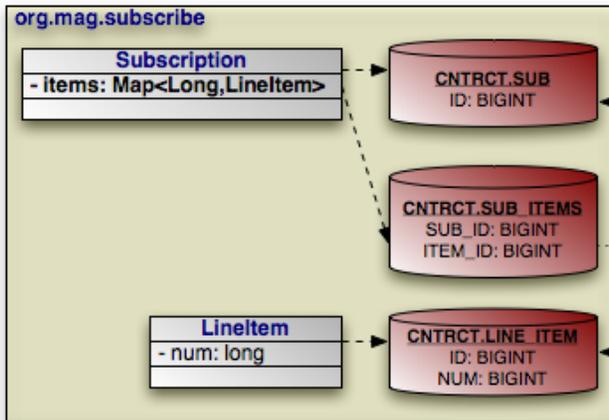
12.8.6. Bidirectional Mapping

Section 5.2.9.1, “Bidirectional Relations” [39] introduced bidirectional relations. To map a bidirectional relation, you map one field normally using the annotations we have covered throughout this chapter. Then you use the `mappedBy` property of the other field’s metadata annotation or the corresponding `mapped-by` XML attribute to refer to the mapped field. Look for this pattern in these bidirectional relations as you peruse the complete mappings below:

- `Magazine.publisher` and `Company.ags`.
- `Article.authors` and `Author.articles`.

12.8.7. Map Mapping

All map fields in EJB persistence are modeled on either one to many or many to many associations. The map key is always derived from an associated entity’s field. Thus map fields use the same mappings as any one to many or many to many fields, namely dedicated **join tables** or **bidirectional relations**. The only additions are the `MapKey` annotation and `map-key` element to declare the key field. We covered these additions in in Section 5.2.13, “Map Key” [41].



The example below maps Subscription's map of LineItems to the SUB_ITEMS join table. The key for each map entry is the LineItem's num field value.

Example 12.16. Join Table Map Mapping

```
package org.mag.subscribe;

@Entity
@Table(name="SUB", schema="CNTRCT")
public class Subscription
{
    @OneToMany(cascade={CascadeType.PERSIST,CascadeType.REMOVE})
    @MapKey(name="num")
    @JoinTable(name="SUB_ITEMS", schema="CNTRCT",
        joinColumns=@JoinColumn(name="SUB_ID"),
        inverseJoinColumns=@JoinColumn(name="ITEM_ID"))
    private Map<Long,LineItem> items;

    ...

    @Entity
    @Table(name="LINE_ITEM", schema="CNTRCT")
    public static class LineItem
        extends Contract
    {
        private long num;

        ...
    }
}
```

The same metadata expressed in XML:

```
<entity class="org.mag.subscribe.Subscription">
  <table name="SUB" schema="CNTRCT"/>
  <attributes>
    ...
    <one-to-many name="items">
      <map-key name="num">
        <join-table name="MAG_ARTS">
          <join-column name="MAG_ISBN" referenced-column-name="ISBN"/>
          <join-column name="MAG_TITLE" referenced-column-name="TITLE"/>
        </join-table>
        <cascade>
          <cascade-persist/>
          <cascade-remove/>
        </cascade>
      </one-to-many>
    ...
  </attributes>
</entity>
<entity class="org.mag.subscribe.Subscription.LineItem">
```

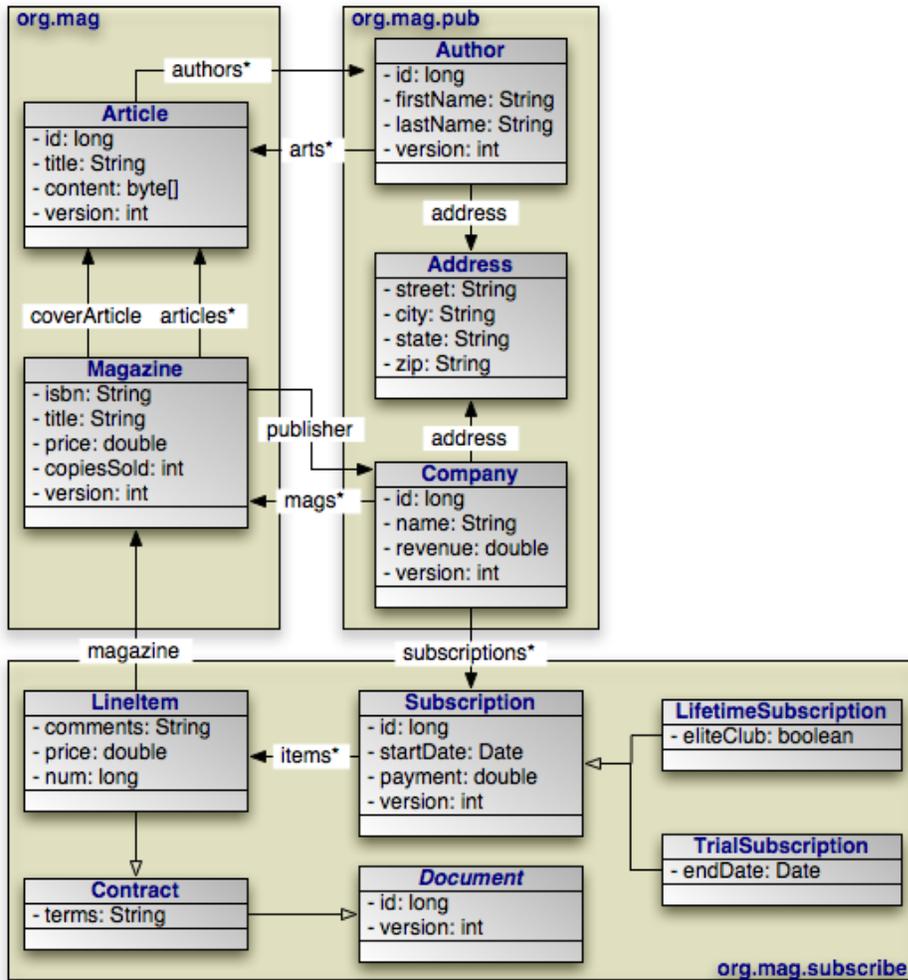
```

<table name="LINE_ITEM" schema="CNTRCT" />
<attributes>
  ..
  <basic name="num"/>
  ..
</attributes>
</entity>

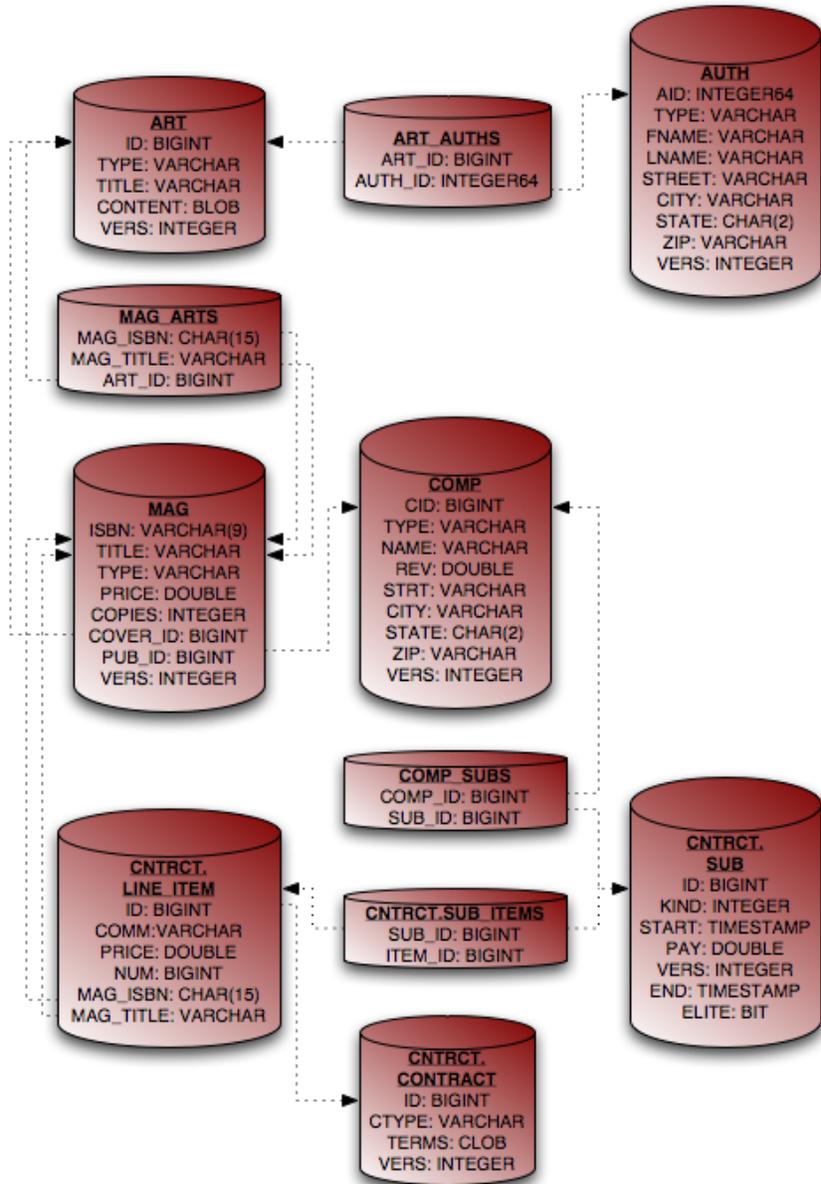
```

12.9. The Complete Mappings

We began this chapter with the goal of mapping the following object model:



That goal has now been met. In the course of explaining EJB's object-relational mapping metadata, we slowly built the requisite schema and mappings for the complete model. First, the database schema:



And finally, the complete entity mappings. We have trimmed the mappings to take advantage of EJB defaults where possible.

Example 12.17. Full Entity Mappings

```

package org.mag;

@Entity
@IdClass(Magazine.MagazineId.class)
@Table(name="MAG")
@DiscriminatorValue("Mag")
public class Magazine
{
    @Column(length=9)
    @Id private String isbn;
    @Id private String title;

    @Column(name="VERS")
    @Version private int version;
}
    
```

```

private String name;
private double price;

@Column(name="COPIES")
private int copiesSold;

@OneToOne(fetch=FetchType.LAZY,
    cascade={CascadeType.PERSIST,CascadeType.REMOVE})
@JoinColumn(name="COVER_ID")
private Article coverArticle;

@OneToMany(cascade={CascadeType.PERSIST,CascadeType.REMOVE})
@OrderBy
@JoinTable(name="MAG_ARTS",
    joinColumns={
        @JoinColumn(name="MAG_ISBN", referencedColumnName="ISBN"),
        @JoinColumn(name="MAG_TITLE", referencedColumnName="TITLE")
    },
    inverseJoinColumns=@JoinColumn(name="ART_ID"))
private Collection<Article> articles;

@ManyToOne(fetch=FetchType.LAZY, cascade=CascadeType.PERSIST)
@JoinColumn(name="PUB_ID")
private Company publisher;

@Transient private byte[] data;

...

public static class MagazineId
{
    ...
}

@Entity
@Table(name="ART", uniqueConstraints=@Unique(columnNames="TITLE"))
@SequenceGenerator(name="ArticleSeq", sequenceName="ART_SEQ")
public class Article
{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="ArticleSeq")
    private long id;

    @Column(name="VERS")
    @Version private int version;

    private String title;
    private byte[] content;

    @ManyToMany(cascade=CascadeType.PERSIST)
    @OrderBy("lastName, firstName")
    @JoinTable(name="ART_AUTHS",
        joinColumns=@JoinColumn(name="ART_ID"),
        inverseJoinColumns=@JoinColumn(name="AUTH_ID"))
    private Collection<Author> authors;

    ...
}

package org.mag.pub;

@Entity
@Table(name="COMP")
public class Company
{
    @Column(name="CID")
    @Id private long id;

    @Column(name="VERS")
    @Version private int version;

    private String name;

    @Column(name="REV")
    private double revenue;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="street", column=@Column(name="STRT")),
        @AttributeOverride(name="city", column=@Column(name="ACITY"))
    })
    private Address address;

    @OneToMany(mappedBy="publisher", cascade=CascadeType.PERSIST)
    private Collection<Magazine> mags;

    @OneToMany(cascade=CascadeType.PERSIST,CascadeType.REMOVE)
    @JoinTable(name="COMP_SUBS",
        joinColumns=@JoinColumn(name="COMP_ID"),
        inverseJoinColumns=@JoinColumn(name="SUB_ID"))

```

```

        private Collection<Subscription> subscriptions;
    }
    ...
}
@Entity
@Table(name="AUTH")
public class Author
{
    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="AuthorGen")
    @TableGenerator(name="AuthorGen", tableName="AUTH_GEN", pkColumnName="PK",
        valueColumnName="AID")
    @Column(name="AID", columnDefinition="INTEGER64")
    private long id;

    @Column(name="VERS")
    @Version private int version;

    @Column(name="FNAME")
    private String firstName;

    @Column(name="LNAME")
    private String lastName;

    private Address address;

    @ManyToMany(mappedBy="authors", cascade=CascadeType.PERSIST)
    private Collection<Article> arts;
}
...
}
@Embeddable
public class Address
{
    private String street;
    private String city;
    @Column(columnDefinition="CHAR(2)")
    private String state;
    private String zip;
}

package org.mag.subscribe;

@MappedSuperclass
public abstract class Document
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

    @Column(name="VERS")
    @Version private int version;
}
...
}
@Entity
@Table(schema="CNTRCT")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="CTYPE")
public class Contract
    extends Document
{
    @Lob
    private String terms;
}
...
}
@Entity
@Table(name="SUB", schema="CNTRCT")
@DiscriminatorColumn(name="KIND", discriminatorType=DiscriminatorType.INTEGER)
@DiscriminatorValue("1")
public class Subscription
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;

    @Column(name="VERS")
    @Version private int version;

    @Column(name="START")
    private Date startDate;

    @Column(name="PAY")
    private double payment;

    @OneToMany(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    @MapKey(name="num")

```

```

@JoinTable(name="SUB_ITEMS", schema="CNRCT",
    joinColumns=@JoinColumn(name="SUB_ID"),
    inverseJoinColumns=@JoinColumn(name="ITEM_ID"))
private Map<Long,LineItem> items;

...

@Entity
@Table(name="LINE_ITEM", schema="CNRCT")
public static class LineItem
    extends Contract
{
    @Column(name="COMM")
    private String comments;

    private double price;
    private long num;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="MAG_ISBN", referencedColumnName="ISBN"),
        @JoinColumn(name="MAG_TITLE", referencedColumnName="TITLE")
    })
    private Magazine magazine;
    ...
}

@Entity(name="Lifetime")
@DiscriminatorValue("2")
public class LifetimeSubscription
    extends Subscription
{
    @Basic(fetch=FetchType.LAZY)
    @Column(name="ELITE")
    private boolean getEliteClub () { ... }
    public void setEliteClub (boolean elite) { ... }
    ...
}

@Entity(name="Trial")
@DiscriminatorValue("3")
public class TrialSubscription
    extends Subscription
{
    @Column(name="END")
    public Date getEndDate () { ... }
    public void setEndDate (Date end) { ... }
    ...
}

```

The same metadata expressed in XML form:

```

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
    version="1.0">
    <mapped-superclass class="org.mag.subscribe.Document">
        <attributes>
            <id name="id">
                <generated-value strategy="IDENTITY"/>
            </id>
            <version name="version">
                <column name="VERS"/>
            </version>
        </attributes>
    </mapped-superclass>
    <entity class="org.mag.Magazine">
        <table name="MAG"/>
        <id-class="org.mag.Magazine.MagazineId"/>
        <discriminator-value>Mag</discriminator-value>
        <attributes>
            <id name="isbn">
                <column length="9"/>
            </id>
            <id name="title"/>
            <basic name="name"/>
            <basic name="price"/>
            <basic name="copiesSold">
                <column name="COPIES"/>
            </basic>
            <version name="version">
                <column name="VERS"/>
            </version>
        </attributes>
    </entity>

```

```

    <many-to-one name="publisher" fetch="LAZY">
      <join-column name="PUB_ID"/>
      <cascade>
        <cascade-persist/>
      </cascade>
    </many-to-one>
    <one-to-many name="articles">
      <order-by/>
      <join-table name="MAG_ARTS">
        <join-column name="MAG_ISBN" referenced-column-name="ISBN"/>
        <join-column name="MAG_TITLE" referenced-column-name="TITLE"/>
        <inverse-join-column name="ART_ID"/>
      </join-table>
      <cascade>
        <cascade-persist/>
        <cascade-remove/>
      </cascade>
    </one-to-many>
    <one-to-one name="coverArticle" fetch="LAZY">
      <join-column name="COVER_ID"/>
      <cascade>
        <cascade-persist/>
        <cascade-remove/>
      </cascade>
    </one-to-one>
    <transient name="data"/>
  </attributes>
</entity>
<entity class="org.mag.Article">
  <table name="ART">
    <unique-constraint>
      <column-name>TITLE</column-name>
    </unique-constraint>
  </table>
  <sequence-generator name="ArticleSeq", sequenceName="ART_SEQ"/>
  <attributes>
    <id name="id">
      <generated-value strategy="SEQUENCE" generator="ArticleSeq"/>
    </id>
    <basic name="title"/>
    <basic name="content"/>
    <version name="version">
      <column name="VERS"/>
    </version>
    <many-to-many name="articles">
      <order-by>lastName, firstName</order-by>
      <join-table name="ART_AUTHS">
        <join-column name="ART_ID" referenced-column-name="ID"/>
        <inverse-join-column name="AUTH_ID" referenced-column-name="AID"/>
      </join-table>
    </many-to-many>
  </attributes>
</entity>
<entity class="org.mag.pub.Company">
  <table name="COMP"/>
  <attributes>
    <id name="id">
      <column name="CID"/>
    </id>
    <basic name="name"/>
    <basic name="revenue">
      <column name="REV"/>
    </basic>
    <version name="version">
      <column name="VERS"/>
    </version>
    <one-to-many name="mags" mapped-by="publisher">
      <cascade>
        <cascade-persist/>
      </cascade>
    </one-to-many>
    <one-to-many name="subscriptions">
      <join-table name="COMP_SUBS">
        <join-column name="COMP_ID"/>
        <inverse-join-column name="SUB_ID"/>
      </join-table>
      <cascade>
        <cascade-persist/>
        <cascade-remove/>
      </cascade>
    </one-to-many>
    <embedded name="address">
      <attribute-override name="street">
        <column name="STRT"/>
      </attribute-override>
      <attribute-override name="city">
        <column name="ACITY"/>
      </attribute-override>
    </embedded>
  </attributes>
</entity>
<entity class="org.mag.pub.Author">
  <table name="AUTH"/>

```

```

<attributes>
  <id name="id">
    <column name="AID" column-definition="INTEGER64" />
    <generated-value strategy="TABLE" generator="AuthorGen" />
    <table-generator name="AuthorGen" table="AUTH_GEN"
      pk-column-name="PK" value-column-name="AID" />
  </id>
  <basic name="firstName">
    <column name="FNAME" />
  </basic>
  <basic name="lastName">
    <column name="LNAME" />
  </basic>
  <version name="version">
    <column name="VERS" />
  </version>
  <many-to-many name="arts" mapped-by="authors">
    <cascade>
      <cascade-persist />
    </cascade>
  </many-to-many>
  <embedded name="address" />
</attributes>
</entity>
<entity class="org.mag.subscribe.Contract">
  <table schema="CNTRCT" />
  <inheritance strategy="JOINED" />
  <discriminator-column name="CTYPE" />
  <attributes>
    <basic name="terms">
      <lob />
    </basic>
  </attributes>
</entity>
<entity class="org.mag.subscribe.Subscription">
  <table name="SUB" schema="CNTRCT" />
  <inheritance strategy="SINGLE_TABLE" />
  <discriminator-value>1</discriminator-value>
  <discriminator-column name="KIND" discriminator-type="INTEGER" />
  <attributes>
    <id name="id">
      <generated-value strategy="IDENTITY" />
    </id>
    <basic name="payment">
      <column name="PAY" />
    </basic>
    <basic name="startDate">
      <column name="START" />
    </basic>
    <version name="version">
      <column name="VERS" />
    </version>
    <one-to-many name="items">
      <map-key name="num">
        <join-table name="SUB_ITEMS" schema="CNTRCT">
          <join-column name="SUB_ID" />
          <inverse-join-column name="ITEM_ID" />
        </join-table>
        <cascade>
          <cascade-persist />
          <cascade-remove />
        </cascade>
      </one-to-many>
    </attributes>
  </entity>
  <entity class="org.mag.subscribe.Subscription.LineItem">
    <table name="LINE_ITEM" schema="CNTRCT" />
    <attributes>
      <basic name="comments">
        <column name="COMM" />
      </basic>
      <basic name="price" />
      <basic name="num" />
      <many-to-one name="magazine">
        <join-column name="MAG_ISBN" referenced-column-name="ISBN" />
        <join-column name="MAG_TITLE" referenced-column-name="TITLE" />
      </many-to-one>
    </attributes>
  </entity>
  <entity class="org.mag.subscribe.LifetimeSubscription" name="Lifetime">
    <discriminator-value>2</discriminator-value>
    <attributes>
      <basic name="eliteClub" fetch="LAZY">
        <column name="ELITE" />
      </basic>
    </attributes>
  </entity>
  <entity class="org.mag.subscribe.TrialSubscription" name="Trial">
    <discriminator-value>3</discriminator-value>
    <attributes>
      <basic name="endDate">
        <column name="END" />
      </basic>
    </attributes>
  </entity>

```

```
    </attributes>
  </entity>
  <embeddable class="org.mag.pub.Address">
    <attributes>
      <basic name="street"/>
      <basic name="city"/>
      <basic name="state">
        <column column-definition="CHAR(2)"/>
      </basic>
      <basic name="zip"/>
    </attributes>
  </embeddable>
</entity-mappings>
```

Chapter 13. Conclusion

This concludes our overview of the EJB persistence specification. The **Kodo EJB Tutorials** continue your EJB education with step-by-step instructions for building simple EJB persistence applications. The **Kodo Reference Guide** contains detailed documentation on all aspects of the Kodo EJB persistence implementation and core development tools.

Part 3. Java Data Objects

Table of Contents

1. Introduction	173
1.1. Intended Audience	173
1.2. Transparent Persistence	173
2. Why JDO?	174
3. JDO Architecture	176
3.1. JDO Exceptions	177
4. PersistenceCapable	179
4.1. Enhancer	179
4.2. Persistence-Capable vs. Persistence-Aware	180
4.3. Restrictions on Persistent Classes	180
4.3.1. Default or No-Arg Constructor	180
4.3.2. Inheritance	180
4.3.3. Persistent Fields	181
4.3.4. Conclusions	183
4.4. Lifecycle Callbacks	183
4.4.1. InstanceCallbacks	183
4.4.2. InstanceLifecycleListener	185
4.5. JDO Identity	186
4.5.1. Datastore Identity	187
4.5.2. Application Identity	187
4.5.2.1. Application Identity Hierarchies	190
4.5.3. Single Field Identity	190
4.6. Conclusions	192
5. Metadata	193
5.1. Persistence Metadata DTD	193
5.2. JDO, Package, and Extension Elements	194
5.3. Class Element	195
5.4. Field Element	196
5.5. Fetch Group Element	198
5.6. The Complete Document	199
5.7. Metadata Placement	200
6. JDOHelper	201
6.1. Persistence-Capable Operations	201
6.2. Lifecycle Operations	202
6.3. PersistenceManagerFactory Construction	205
7. PersistenceManagerFactory	207
7.1. Obtaining a PersistenceManagerFactory	207
7.2. PersistenceManagerFactory Properties	208
7.2.1. Connection Configuration	208
7.2.2. PersistenceManager and Transaction Defaults	209
7.3. Obtaining PersistenceManagers	211
7.4. Properties and Supported Options	211
7.5. DataStoreCache Access	213
7.6. Closing the PersistenceManagerFactory	213
8. PersistenceManager	214
8.1. User Object Association	215
8.2. Configuration Properties	215
8.3. Transaction Association	216
8.4. FetchPlan Association	216
8.5. Persistence-Capable Lifecycle Management	216
8.6. Lifecycle Examples	218
8.7. Detach and Attach Functionality	219
8.8. JDO Identity Management	221

8.9. Cache Management	222
8.10. Extent Factory	223
8.11. Query Factory	223
8.12. Sequence Factory	224
8.13. Connection Access	224
8.14. Closing	225
9. Transaction	226
9.1. Transaction Types	226
9.2. The JDO Transaction Interface	227
9.2.1. Transaction Properties	227
9.2.2. Transaction Demarcation	228
10. Extent	230
11. Query	232
11.1. Object Filtering	232
11.2. JDOQL	234
11.3. Advanced Object Filtering	236
11.4. Compiling and Executing Queries	239
11.5. Limits and Ordering	240
11.6. Projections	241
11.7. Aggregates	244
11.8. Result Class	246
11.8.1. JavaBean Result Class	246
11.8.2. Generic Result Class	248
11.9. Single-String JDOQL	248
11.10. Named Queries	250
11.10.1. Defining Named Queries	250
11.10.2. Executing Named Queries	252
11.11. Delete By Query	252
11.12. Conclusion	253
12. FetchPlan	254
12.1. Detachment Options	255
13. DataStoreCache	257
14. JDOR	259
15. Mapping Metadata	260
15.1. Mapping Metadata Placement	260
15.2. Mapping Metadata DTD	261
15.3. Sequences	263
15.4. Class Table	264
15.5. Datastore Identity	266
15.6. Column	268
15.7. Joins	269
15.7.1. Shortcuts	270
15.7.2. Self Joins	271
15.7.3. Target Fields	272
15.8. Inheritance	273
15.8.1. subclass-table	273
15.8.1.1. Advantages	274
15.8.1.2. Disadvantages	274
15.8.1.3. Additional Considerations	274
15.8.2. new-table	275
15.8.2.1. Joined	275
15.8.2.1.1. Advantages	277
15.8.2.1.2. Disadvantages	277
15.8.2.2. Table Per Class	277
15.8.2.2.1. Advantages	278
15.8.2.2.2. Disadvantages	278
15.8.3. superclass-table	279
15.8.3.1. Advantages	279
15.8.3.2. Disadvantages	279

15.8.4. Putting it All Together	279
15.9. Discriminator	281
15.9.1. class-name	281
15.9.2. value-map	282
15.9.3. none	282
15.9.4. Putting it All Together	282
15.10. Version	284
15.10.1. none	285
15.10.2. version-number	285
15.10.3. date-time	285
15.10.4. state-comparison	285
15.10.5. Putting it All Together	286
15.11. Field Mapping	287
15.11.1. Superclass Fields	288
15.11.2. Basic Mapping	288
15.11.2.1. CLOB	290
15.11.2.2. BLOB	291
15.11.3. Automatic Values	291
15.11.4. Secondary Tables	292
15.11.5. Direct Relations	294
15.11.5.1. Inverse Keys	295
15.11.5.2. Bidirectional Relations	296
15.11.6. Basic Collections	297
15.11.7. Association Table Collections	298
15.11.7.1. Bidirectional Relations	301
15.11.8. Inverse Key Collections	301
15.11.8.1. Bidirectional Relations	303
15.11.9. Maps	304
15.11.10. Embedded Objects	306
15.12. Foreign Keys	309
15.13. Indexes	311
15.14. Unique Constraints	312
15.15. The Complete Document	313
16. Sequence	318
17. SQL Queries	320
17.1. Creating SQL Queries	320
17.2. Retrieving Persistent Objects with SQL	321
17.3. SQL Projections	322
17.4. Named SQL Queries	324
17.5. Conclusion	324
18. Conclusion	325

Chapter 1. Introduction

Java Data Objects (JDO) is a specification from Sun Microsystems for the transparent persistence of Java objects to any transactional datastore. This document provides an overview of JDO. The information presented applies to all JDO implementations, unless otherwise noted.

Note

For coverage of Kodo's many extensions to the JDO specification, see the [Reference Guide](#).

1.1. Intended Audience

This document is intended for developers who want to learn about JDO in order to use it in their applications. It assumes that you have a strong knowledge of Java and object-oriented concepts, and a familiarity with the eXtensible Markup Language (XML). This document does *not*, however, assume any experience with database programming or the manipulation of persistent data in general.

1.2. Transparent Persistence

Persistent data is information that can outlive the program that creates it. The majority of complex programs use persistent data: GUI applications need to store user preferences across program invocations, web applications track user movements and orders over long periods of time, etc.

Transparent persistence is the storage and retrieval of persistent data with little or no work from you, the developer. For example, Java serialization is a form of transparent persistence because it can be used to persist Java objects directly to a file with very little effort. Serialization's capabilities as a transparent persistence mechanism pale in comparison to those provided by JDO, however. The next chapter compares JDO to serialization and other available persistence mechanisms.

Chapter 2. Why JDO?

Java developers who need to store and retrieve persistent data already have several options available to them: serialization, JDBC, object-relational mapping tools, object databases, EJB 2 entities, and JPA. Why introduce yet another persistence framework? The answer to this question is that each of the aforementioned persistence solutions has severe limitations. JDO attempts to overcome these limitations, as illustrated by the table below.

Table 2.1. Persistence Mechanisms

Supports:	Serialization	JDBC	ORM	ODB	EJB 2	JPA	JDO
Java Objects	Yes	No	Yes	Yes	Yes	Yes	Yes
Advanced OO Concepts	Yes	No	Yes	Yes	No	Yes	Yes
Transactional Integrity	No	Yes	Yes	Yes	Yes	Yes	Yes
Concurrency	No	Yes	Yes	Yes	Yes	Yes	Yes
Large Data Sets	No	Yes	Yes	Yes	Yes	Yes	Yes
Existing Schema	No	Yes	Yes	No	Yes	Yes	Yes
Relational and Non-Relational Stores	No	No	No	No	Yes	No	Yes
Queries	No	Yes	Yes	Yes	Yes	Yes	Yes
Strict Standards / Portability	Yes	No	No	No	Yes	Yes	Yes
Simplicity	Yes	Yes	Yes	Yes	No	Yes	Yes

- *Serialization* is Java's built-in mechanism for transforming an object graph into a series of bytes, which can then be sent over the network or stored in a file. Serialization is very easy to use, but it is also very limited. It must store and retrieve the entire object graph at once, making it unsuitable for dealing with large amounts of data. It cannot undo changes that are made to objects if an error occurs while updating information, making it unsuitable for applications that require strict data integrity. Multiple threads or programs cannot read and write the same serialized data concurrently without conflicting with each other. It provides no query capabilities. All these factors make serialization useless for all but the most trivial persistence needs.
- Many developers use the *Java Database Connectivity* (JDBC) APIs to manipulate persistent data in relational databases. JDBC overcomes most of the shortcomings of serialization: it can handle large amounts of data, has mechanisms to ensure data integrity, supports concurrent access to information, and has a sophisticated query language in SQL. Unfortunately, JDBC does not duplicate serialization's ease of use. The relational paradigm used by JDBC was not designed for storing objects, and therefore forces you to either abandon object-oriented programming for the portions of your code that deal with persistent data, or to find a way of mapping object-oriented concepts like inheritance to relational databases yourself.
- There are many proprietary software products that can perform the mapping between objects and relational database tables for you. These *object-relational mapping* (ORM) frameworks allow you to focus on the object model and not concern yourself with the mismatch between the object-oriented and relational paradigms. Unfortunately, each of these product has its own set of APIs. Your code becomes tied to the proprietary interfaces of a single vendor. If the vendor raises prices, fails to fix show-stopping bugs, or falls behind in features, you cannot switch to another product without rewriting all of your persistence code. This is referred to as vendor lock-in.

- Rather than map objects to relational databases, some software companies have developed a form of database designed specifically to store objects. These *object databases* (ODBs) are often much easier to use than object-relational mapping software. The Object Database Management Group (ODMG) was formed to create a standard API for accessing object databases; few object database vendors, however, comply with the ODMG's recommendations. Thus, vendor lock-in plagues object databases as well. Many companies are also hesitant to switch from tried-and-true relational systems to the relatively unknown object database technology. Fewer data-analysis tools are available for object database systems, and there are vast quantities of data already stored in older relational databases. For all of these reasons and more, object databases have not caught on as well as their creators hoped.
- The Enterprise Edition of the Java platform introduced entity Enterprise Java Beans (EJBs). Entity EJBs are components that represent persistent information in a datastore. Like object-relational mapping solutions, entity EJBs provide an object-oriented view of persistent data. Unlike object-relational software, however, entity EJBs are not limited to relational databases; the persistent information they represent may come from an Enterprise Information System (EIS) or other storage device. Also, EJBs use a strict standard, making them portable across vendors. Unfortunately, the EJB standard is somewhat limited in the object-oriented concepts it can represent. Advanced features like inheritance, polymorphism, and complex relations are absent. Additionally, EJBs are difficult to code, and they require heavyweight and often expensive application servers to run. EJBs, especially session and message-driven beans, do have other advantages, however, and so the JDO specification details how to integrate JDO and EJBs.
- The upcoming EJB 3 specification introduces the Java Persistence API (JPA). This API is strikingly similar to JDO, and is much easier to use than the EJB 2 programming model. JPA does not, however, support non-relational databases.

JDO combines the best features from each of the persistence mechanisms listed above. Creating persistent classes under JDO is as simple as creating serializable classes. JDO supports the large data sets, data consistency, concurrent use, and query capabilities of JDBC. Like object-relational software and object databases, JDO allows the use of advanced object-oriented concepts such as inheritance. JDO avoids vendor lock-in by relying on a strict specification like entity EJBs. Like entities in EJB 2, JDO does not prescribe any specific back-end datastore. JDO implementations might store objects in relational databases, object databases, flat files, or any other persistent storage device. And like JPA, JDO is extremely easy to use.

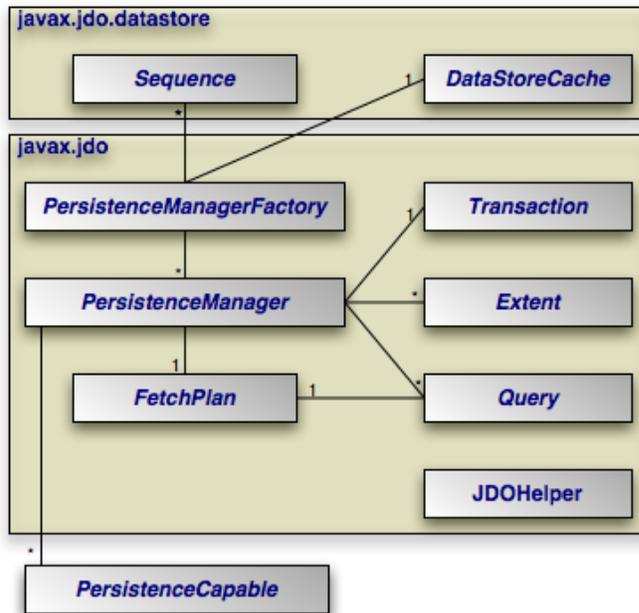
Note

By default, Kodo stores objects in relational databases using JDBC. It can be customized for use with other datastores.

JDO is not ideal for every application. For many applications, though, it provides an exciting alternative to other persistence mechanisms.

Chapter 3. JDO Architecture

The diagram below illustrates the relationships between the primary components of the JDO architecture.



- **JDOHelper:** The `javax.jdo.JDOHelper` contains static helper methods to query the lifecycle state of persistent objects and to obtain `PersistenceManagerFactory` instances in a vendor-neutral fashion.
- **PersistenceManagerFactory:** The `javax.jdo.PersistenceManagerFactory` is a factory for `PersistenceManagers`.
- **PersistenceManager:** The `javax.jdo.PersistenceManager` is the primary JDO interface used by applications. Each `PersistenceManager` manages a set of persistent objects, and has APIs to insert new objects and delete existing ones. There is a one-to-one relationship between a `PersistenceManager` and a `Transaction`. `PersistenceManagers` also act as factories for `Extent` and `Query` instances.
- **PersistenceCapable:** JDO calls user-defined persistent classes *persistence-capable* classes. Most JDO vendors provide an *enhancer* to transparently add a special `PersistenceCapable` interface to each persistent class. You will never use this interface directly.
- **Transaction:** Each `PersistenceManager` has a one-to-one relation with a single `javax.jdo.Transaction`. Transactions allow operations on persistent data to be grouped into units of work that either completely succeed or completely fail, leaving the datastore in its original state. These all-or-nothing operations are important for maintaining data integrity.
- **Extent:** A `javax.jdo.Extent` is a logical view of all the objects of a particular class that exist in the datastore. You can configure `Extents` to also include subclasses. `Extents` are obtained from a `PersistenceManager`.
- **Query:** The `javax.jdo.Query` interface is implemented by each JDO vendor to find persistent objects that meet certain criteria. JDO standardizes support for queries using the Java Data Objects Query Language (JDOQL), and for relational database queries using the Structured Query Language (SQL). You obtain `Query` instances from a `PersistenceManager`.
- **FetchPlan:** Each `PersistenceManager` and `Query` has a mutable reference to a `javax.jdo.FetchPlan`. The `FetchPlan` gives you control over eager fetching, result scrolling, and other data loading behavior.

- **Sequence:** The `javax.jdo.datastore.Sequence` interface represents an object capable of generating sequential values. Sequences are defined in mapping metadata (see [Chapter 15, Mapping Metadata \[260\]](#)) cached at the `PersistenceManagerFactory` level, and obtained through the `PersistenceManager`.
- **DataStoreCache:** Many JDO implementations provide some form of datastore cache. The `javax.jdo.datastore.DataStoreCache` interface provides a standard way to interact with your vendor's cache.

The example below illustrates how the JDO interfaces interact to execute a JDOQL query and update persistent objects.

Example 3.1. Interaction of JDO Interfaces

```
// get a PersistenceManagerFactory using the JDOHelper; typically
// the factory is cached for easy repeated use
PersistenceManagerFactory factory =
    JDOHelper.getPersistenceManagerFactory (System.getProperties ());

// get a PersistenceManager from the factory
PersistenceManager pm = factory.getPersistenceManager ();

// updates take place within transactions
Transaction tx = pm.currentTransaction ();
tx.begin ();

// query for all employees who work in our research division
// and put in over 40 hours a week average
Extent extent = pm.getExtent (Employee.class, false);
Query query = pm.newQuery (extent);
query.setFilter ("division.name == 'Research' && avgHours > 40");

// we only need to populate the metadata-defined salary fetch group data;
// there might be a lot of employees, so read from store in batches of 100
query.getFetchPlan ().setGroup ("salary");
query.getFetchPlan ().setFetchSize (100);
List results = (List) query.execute ();

// give all those hard-working employees a raise; keep track of hardest worker
Employee emp;
Employee mostHours = null;
for (Iterator itr = results.iterator (); itr.hasNext ());
{
    emp = (Employee) itr.next ();
    emp.setSalary (emp.getSalary () * 1.1);
    if (mostHours == null || emp.avgHours () > mostHours.avgHours ())
        mostHours = emp;
}

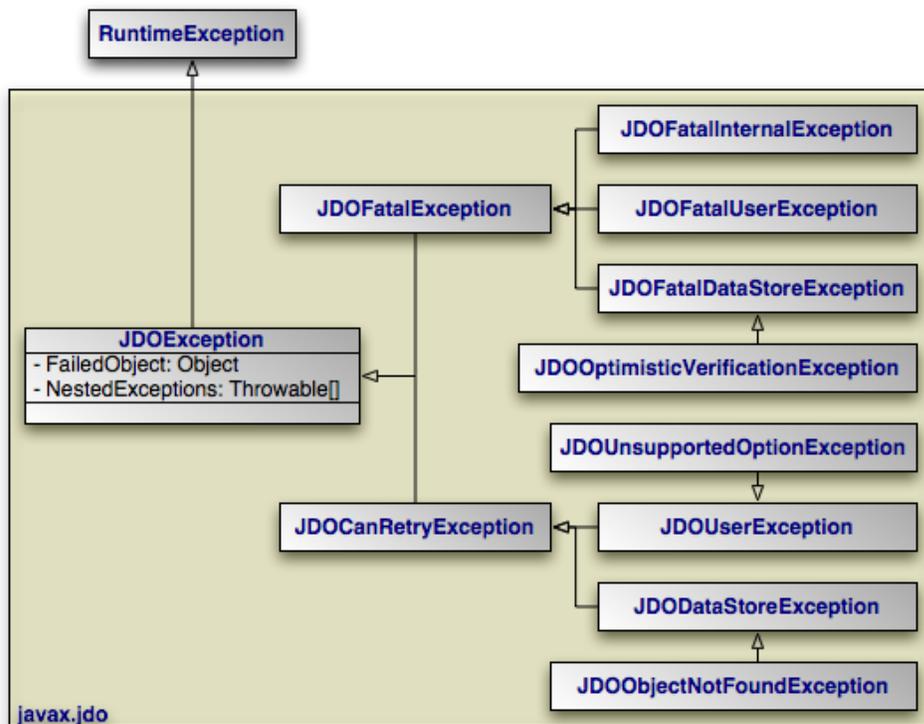
// we know we're going to be fetching the hardest-working employee a
// lot in our application, so pin its data to the cache
factory.getDataStoreCache ().pin (JDOHelper.getObjectId (mostHours));

// commit the updates and free persistence manager
tx.commit ();
pm.close ();

// if we were ending our application, we'd free the factory too
// factory.close ();
```

The remainder of this document explores the JDO interfaces in detail. We present them in roughly the order that you will use them as you develop your application.

3.1. JDO Exceptions



The diagram above depicts the JDO exception architecture. Runtime exceptions such as `NullPointerException` and `IllegalArgumentException` aside, JDO components only throw **JDOExceptions**.

The JDO exception hierarchy should be self-explanatory. The base `JDOException` class provides the following useful properties:

- **FailedObject**: The failed object is the persistent instance or identity object that caused the exception, if applicable. It is particularly useful for `JDOOptimisticVerificationExceptions` and `JDOObjectNotFoundExceptions`.
- **NestedExceptions**: An array of nested exceptions that caused the failure. Rather than stop immediately at the first error, many JDO methods collect all errors during execution and throw a single parent exception, nesting the collected exceptions within it. For example, a failed attempt to commit a transaction might result in a `JDOFatalDataStoreException` with a nested `JDOOptimisticVerificationException` for each persistent instance that failed the optimistic concurrency check.

See the **Javadoc** for additional details on JDO exceptions.

Chapter 4. PersistenceCapable

In most JDO implementations, user-defined persistent classes implement a special `PersistenceCapable` interface. This interface contains many complex methods that enable the JDO implementation to manage the persistent fields of class instances. Fortunately, you do not have to implement this interface yourself. In fact, writing a persistent class in JDO is usually no different than writing any other class. There are no special parent classes to extend from, field types to use, or methods to write. This is one important way in which JDO makes persistence completely transparent to you, the developer.

Example 4.1. PersistenceCapable Class

```
package org.mag;

/**
 * Example persistent class. Notice that it looks exactly like any other
 * class. JDO makes writing persistent classes completely transparent.
 */
public class Magazine
{
    private String    isbn;
    private String    title;
    private Set       articles = new HashSet ();
    private Article   coverArticle;
    private int       copiesSold;
    private double    price;
    private Company   publisher;

    private Magazine ()
    {
    }

    public Magazine (String title, String isbn)
    {
        this.title = title;
        this.isbn = isbn;
    }

    public void publish (Company publisher, double price)
    {
        this.publisher = publisher;
        publisher.addMagazine (this);
        this.price = price;
    }

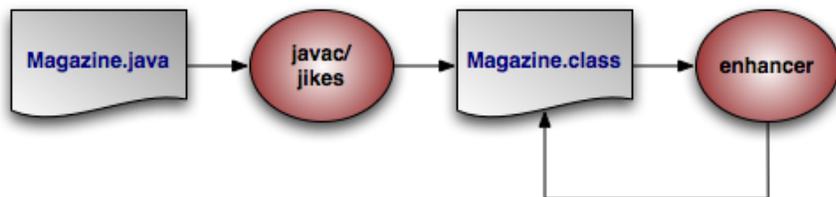
    public void sell ()
    {
        copiesSold++;
        publisher.addRevenue (price);
    }

    public void addArticle (Article article)
    {
        articles.add (article);
    }

    // rest of methods omitted
}
```

4.1. Enhancer

In order to shield you from the intricacies of the `PersistenceCapable` interface, JDO implementations typically provide an *enhancer*. An enhancer is a tool that automatically adds code to your persistent classes after you have written them. Though some vendors may use source enhancers that modify your Java code, enhancers generally operate on `.class` files. They post-process the bytecode generated by your Java compiler, adding the necessary fields and methods to implement the expected interface. This bytecode modification perfectly preserves the line numbers in stack traces and is compatible with Java debuggers, so enhancement does not affect debugging.



The diagram above illustrates the compilation of a persistent class. JDO implementations typically include an Ant task so that you can make enhancement an automatic part of your build process. Some JDO implementations may also use custom class loaders or Java 5's class loading hooks to perform enhancement transparently at runtime, rather than adding a step to the build process.

Note

Kodo uses bytecode, rather than source code, enhancement. Kodo offers both a compile-time enhancer tool and a runtime enhancement option for Java 5 users. See [Section 5.2, “Enhancement” \[452\]](#) of the Reference Guide.

4.2. Persistence-Capable vs. Persistence-Aware

Classes that have been enhanced to implement the `PersistenceCapable` interface are referred to as *persistence-capable* classes. Classes that directly access public or protected persistent fields of persistence-capable classes are called *persistence-aware*. Persistence-aware classes must also be enhanced - each time a persistence-aware class directly accesses a persistent field of a persistence-capable class, the enhancer adds code to notify the JDO implementation that the field in question is about to be read or written. This enables the JDO implementation to synchronize the field's value with the datastore as needed. Unless the persistence-aware class is also persistence-capable, the enhancer does not add code to make the class implement the `PersistenceCapable` interface.

Generally, it is best to keep all of your persistent fields private, or protected but only accessed by persistent subclasses. In addition to the standard arguments in favor of state encapsulation, this approach avoids the hassle of tracking which non-persistent classes must be enhanced as persistence-aware because they happen to access a public or protected field of some persistent class.

4.3. Restrictions on Persistent Classes

There are very few restrictions placed on persistent classes. Still, it never hurts to familiarize yourself with exactly what JDO does and does not support.

4.3.1. Default or No-Arg Constructor

The JDO specification requires that all persistence-capable classes must have a no-arg constructor. This constructor may be private. Because the compiler automatically creates a default no-arg constructor when no other constructor is defined, only classes that define constructors must also include a no-arg constructor.

Note

Kodo's enhancer will automatically add a protected no-arg constructor to your class when required. Therefore, this restriction does not apply under Kodo.

4.3.2. Inheritance

JDO fully supports inheritance in persistent classes. It allows persistent classes to inherit from non-persistent classes, persistent classes to inherit from other persistent classes, and non-persistent classes to inherit from persistent classes. It is even possible to form inheritance hierarchies in which persistence skips generations. There are, however, a few important limitations:

- Persistent classes cannot inherit from certain natively-implemented system classes such as `java.net.Socket` and `java.lang.Thread`.
- If a persistent class inherits from a non-persistent class, the fields of the non-persistent superclass cannot be persisted.
- All classes in an inheritance tree must use the same JDO identity type. If they use application identity, they must either use the same identity class, or else they must each declare that they use separate identity classes whose inheritance hierarchy exactly mirrors the inheritance hierarchy of the persistent class hierarchy. We cover JDO identity in **Section 4.5, “JDO Identity” [186]**

4.3.3. Persistent Fields

JDO manages the state of all persistent fields. Before you access a field, the JDO runtime makes sure that it has been loaded from the datastore. When you set a field, the runtime records that it has changed so that the new value will be persisted. This allows you to treat the field in exactly the same way you treat any other field - another aspect of JDO's transparent persistence.

JDO does not support static or final fields. It does, however, include built-in support for most common field types. These types can be roughly divided into three categories: immutable types, mutable types, and relations.

Immutable types, once created, cannot be changed. The only way to alter a persistent field of an immutable type is to assign a new value to the field. JDO supports the following immutable types:

- All primitives (`int`, `float`, `byte`, etc)
- All primitive wrappers (`java.lang.Integer`, `java.lang.Float`, `java.lang.Byte`, etc)
- `java.lang.String`
- `java.math.BigInteger`
- `java.math.BigDecimal`
- `java.lang.Number`
- `java.util.Locale`

Persistent fields of *mutable* types can be altered without assigning the field a new value. Mutable types can be modified directly through their own methods. The JDO specification requires that implementations support the following mutable field types:

- `java.util.Date`
- `java.util.Collection`
- `java.util.HashSet`
- `java.util.Map`
- `java.util.HashMap`
- `java.util.Hashtable`

Collection and map types may be parameterized.

Most implementations do not allow you to persist nested mutable types, such as Collections of Maps.

Note

Kodo supports all JDK 1.2 Set, List, and Map types, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, enums, and many other mutable and immutable field types. Kodo also allows you to plug in support for custom types.

Most JDO implementations support mutable fields by transparently replacing the field value with an instance of a special subclass of the field's declared type. For example, if your persistent object has a field containing a `java.util.Date`, the JDO implementation will transparently replace the value of that field at runtime with some vendor-specific `Date` subclass -- call it `JDODate`. The job of this subclass is to track modifications to the field. Thus the `JDODate` class will override all mutator methods of `Date` to notify the JDO implementation that the field's value has been changed. The JDO implementation then knows to write the field's new value to the datastore at the next opportunity.

Of course, when you develop and use persistent classes, this is all transparent. You continue to use the standard methods of mutable fields as you normally would. It is important to know how support for mutable fields is implemented, however, in order to understand why JDO has such trouble with arrays. JDO allows you to use persistent array fields, and it automatically detects when these fields are assigned a new array value or set to `null`. Because arrays cannot be subclassed, however, JDO cannot detect when new values are written to array indexes. If you set an index of a persistent array, you must either reset the array field, or explicitly tell the JDO implementation you have changed it; this is referred to as *dirtying* the field. Dirtying is accomplished through the `JDOHelper`'s `makeDirty` method.

Example 4.2. Accessing Mutable Persistent Fields

```
/**
 * Example demonstrating the use of mutable persistent fields in JDO.
 * Assume Person is a persistent class.
 */
public void addChild (Person parent, Person child)
{
    // can modify most mutable types directly; JDO tracks
    // the modifications for you
    Date lastUp = parent.getLastUpdated ();
    lastUp.setTime (System.currentTimeMillis ());
    Collection children = parent.getChildren ();
    children.add (child);
    child.setParent (parent);

    // arrays need explicit dirtying if they are modified,
    // but not if the field is reset
    parent.setObjectArray (new Object[0]);
    child.getObjectArray ()[0] = parent;
    JDOHelper.makeDirty (child, "objectArray");
    // or: child.setObjectArray (child.getObjectArray ());
}
```

As the parent-child example above illustrates, JDO supports relations between persistent objects in addition to the standard Java types covered so far. All JDO implementations should allow user-defined persistent classes as well as collections and maps of user-defined persistent classes as persistent field types. The exact collection and map types you can use to hold persistent relations will depend on which mutable field types the implementation supports.

Most JDO implementations also have some support for fields whose concrete class is not known. Fields declared as type `java.lang.Object` or as a user-defined interface type fall into this category. Because these fields are so general, though, there may be limitations placed on them. For example, they may be impossible to query, and loading and/or storing them may be inefficient.

Note

Kodo supports user-defined persistent objects as elements of any of the supported collection types. It also supports user-defined persistent objects as keys, values, or both in any supported map type.

Kodo supports persistent `java.lang.Object` fields by serializing the field value and storing it as a sequence of bytes. It supports persistent interface fields by storing the object id of the instance stored in the field, then re-fetching the corresponding object when the field is loaded. Collections and maps where the element/key/value type is `java.lang.Object` or an interface are fully supported as well.

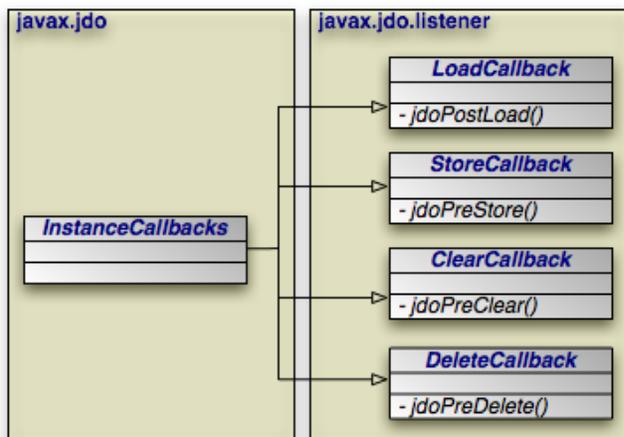
4.3.4. Conclusions

This section detailed all of the restrictions JDO places on persistent classes. While it may seem like we presented a lot of information, you will seldom find yourself hindered by these restrictions in practice. Additionally, there are often ways of using JDO's other features to circumvent any limitations you run into. The next section explores a powerful JDO feature that is particularly useful for this purpose.

4.4. Lifecycle Callbacks

It is often necessary to perform various actions at different stages of a persistent object's lifecycle. JDO includes two mechanisms for monitoring changes in the lifecycle of your persistent objects: the `InstanceCallbacks` interface, and the `InstanceLifecycleListener` event framework.

4.4.1. InstanceCallbacks



Your persistent classes can implement the `InstanceCallbacks` family of interfaces to receive callbacks when certain JDO lifecycle events take place. There are four callbacks available:

- The `LoadCallback.jdoPostLoad` method is called by the JDO implementation after the default fetch group fields of your class have been loaded from the datastore. Default fetch groups are explained in [Chapter 5, Metadata \[193\]](#) for now think of the default fetch group as all of the primitive fields of the object. No other persistent fields can be accessed in this method.

`jdoPostLoad` is often used to initialize non-persistent fields whose values depend on the values of persistent fields. An example of this is presented below.

- `StoreCallback.jdoPreStore` is called just before the persistent values in your object are flushed to the datastore. You can access all persistent fields in this method.

`jdoPreStore` is the complement to `jdoPostLoad`. While `jdoPostLoad` is most often used to initialize non-persistent values from persistent data, `jdoPreStore` is usually used to set persistent fields with information cached in non-persistent ones. See the example below. Note that the persistent identity of the object may not have been assigned yet when this method is called.

- The `ClearCallback.jdoPreClear` method is called before the persistent fields of your object are cleared. JDO implementations clear the persistent state of objects for several reasons, most of which will be covered later in this document. You can use `jdoPreClear` to clear non-persistent cached data and null relations to other objects. You should not access the values of persistent fields in this method.
- `DeleteCallback.jdoPreDelete` is called before an object transitions to the deleted state. Access to persistent fields is valid within this method. You might use this method to cascade the deletion to related objects based on complex criteria, or to perform other cleanup.

Unlike the `PersistenceCapable` interface, you must implement the `InstanceCallbacks` interfaces explicitly if you want to receive lifecycle callbacks.

Note

The `javax.jdo.InstanceCallbacks` interface is mainly present for backwards compatibility with previous JDO versions. You will typically implement the specific `javax.jdo.listener` package interfaces for the callbacks you want to receive. However, it may be convenient to implement `InstanceCallbacks` rather than the individual interfaces when your class utilizes several callbacks.

Example 4.3. Using Callback Interfaces

```
/**
 * Example demonstrating the use of the InstanceCallbacks interface to
 * persist a java.net.InetAddress and implement a privately-owned relation.
 */
public class Host
    implements LoadCallback, StoreCallback, DeleteCallback
{
    // the InetAddress field cannot be persisted directly by JDO, so we
    // use the jdoPostLoad and jdoPreStore methods below to persist it
    // indirectly through its host name string
    private transient InetAddress address; // non-persistent
    private String hostName; // persistent

    // set of devices attached to this host
    private Set devices = new HashSet ();

    // setters, getters, and business logic omitted

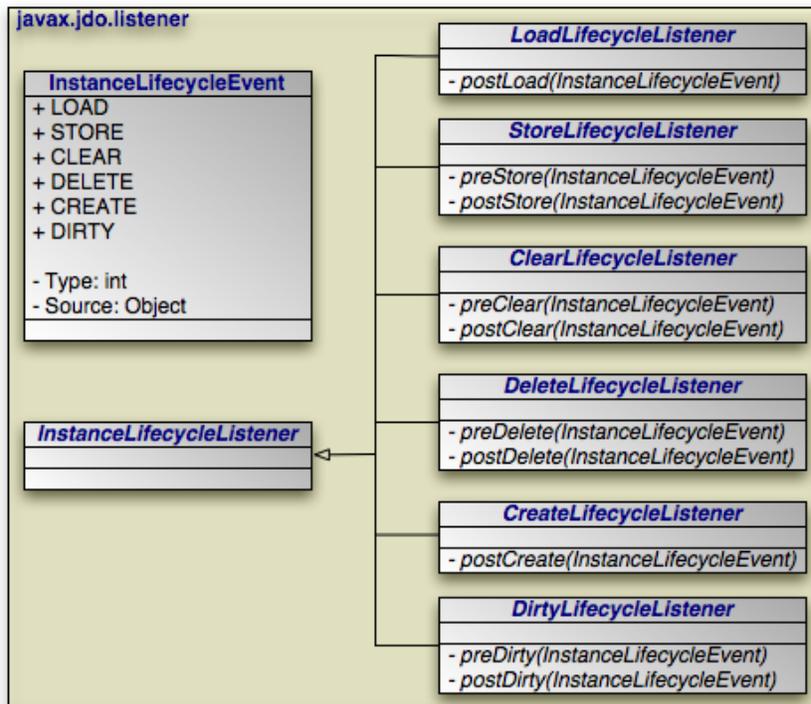
    public void jdoPostLoad ()
    {
        // form the InetAddress using the persistent host name
        try
        {
            address = InetAddress.getByName (hostName);
        }
        catch (IOException ioe)
        {
            throw new JDOException ("Invalid host name: " + hostName, ioe);
        }
    }

    public void jdoPreStore ()
    {
        // store the host name information based on the InetAddress values
        hostName = address.getHostName ();
    }

    public void jdoPreDelete ()
    {
        // delete certain related devices when this object is deleted, based
        // on business logic
        PersistenceManager pm = JDOHelper.getPersistenceManager (this);
        pm.deletePersistentAll (filterDependents (devices));
    }
}
```

```
}
}
```

4.4.2. InstanceLifecycleListener



Note

The `kodo.event` package contains additional Kodo events beyond the JDO standard; see [Section 9.8, “Transaction Events”](#) [559] of the Reference Guide and the package [Javadoc](#) for details.

Only persistent classes can implement the `InstanceCallbacks` interfaces. This makes sense for lifecycle actions such as caching internal state and deleting dependent relations, but is clumsy for cross-cutting concerns like logging and auditing. The lifecycle listener event framework solves this problem by allowing non-persistent classes to subscribe to lifecycle events. The framework consists of a common event class, a common super-interface for event listeners, and several individual listener interfaces. A concrete listener class can implement any combination of listener interfaces.

- **InstanceLifecycleEvent**: The event class. The source of a lifecycle event is the persistent object for which the event was triggered.
- **InstanceLifecycleListener**: Common base interface for all listener types. `InstanceLifecycleListener` has no operations, but gives a measure of type safety when adding listener objects to a `PersistenceManager` or `PersistenceManagerFactory`. See [Chapter 8, *PersistenceManager*](#) [214] and [Chapter 7, *PersistenceManagerFactory*](#) [207]
- **LoadLifecycleListener**: Listens for persistent state loading events. Its `postLoad` method is equivalent to the `InstanceCallbacks.jdoPostLoad` method described above.

- **StoreLifecycleListener**: Listens for persistent state flushes. Its `preStore` method is equivalent to the `InstanceCallbacks.jdoPreStore` method described above. Its `postStore` handler is invoked after the data for the source object has been flushed to the database. Unlike `preStore`, the source object is guaranteed to have a persistent identity by the time `postStore` is triggered.
- **ClearLifecycleListener**: Receives notifications when objects clear their persistent state. Its `preClear` method is equivalent to `InstanceCallbacks.jdoPreClear`. The `postClear` event is sent just after the source object's state is cleared.
- **DeleteLifecycleListener**: Listens for object deletion events. Its `preDelete` method is equivalent to `InstanceCallbacks.jdoPreDelete`. Its `postDelete` handler is triggered after the source object has transitioned to the deleted state. Access to persistent fields is not allowed in `postDelete`.
- **CreateLifecycleListener**: The `postCreate` event is fired when an object first transitions from unmanaged to persistent-new, such as during a call to `PersistenceManager.makePersistent`.
- **DirtyLifecycleListener**: Dirty events fire when an object is first modified (in JDO parlance, becomes dirty) within a transaction. The runtime invokes `preDirty` before applying the change to the object, and `postDirty` after applying the change.

4.5. JDO Identity

Java recognizes two forms of object identity: numeric identity and qualitative identity. If two references are *numerically* identical, then they refer to the same JVM instance in memory. You can test for this using the `==` operator. *Qualitative* identity, on the other hand, relies on some user-defined criteria to determine whether two objects are "equal". You test for qualitative identity using the `equals` method. By default, this method simply relies on numeric identity.

JDO introduces another form of object identity, called JDO identity. JDO identity tests whether two persistent objects represent the same state in the datastore.

The JDO identity of each persistent instance is encapsulated in its *JDO identity object*. You can obtain the JDO identity object for a persistent instance through the **JDOHelper**'s `getObjectId` method. If two JDO identity objects compare equal using the `equals` method, then the two persistent objects represent the same state in the datastore.

Example 4.4. JDO Identity Objects

```
/**
 * This method tests whether the given persistent objects represent the
 * same datastore record. It returns false if either argument is not
 * a persistent object.
 */
public boolean persistentEquals (Object obj1, Object obj2)
{
    Object jdoId1 = JDOHelper.getObjectId (obj1);
    Object jdoId2 = JDOHelper.getObjectId (obj2);
    return jdoId1 != null && jdoId1.equals (jdoId2);
}
```

If you are dealing with a single **PersistenceManager**, then there is an even easier way to test whether two persistent object references represent the same state in the datastore: the `==` operator. JDO requires that each `PersistenceManager` maintain only one JVM object to represent each unique datastore record. Thus, JDO identity is equivalent to numeric identity within a `PersistenceManager`'s cache of managed objects. This is referred to as the *uniqueness requirement*.

The uniqueness requirement is extremely important - without it, it would be impossible to maintain data integrity. Think of what could happen if two different objects of the same `PersistenceManager` were allowed to represent the same persistent data.

If you made different modifications to each of these objects, which set of changes should be written to the datastore? How would your application logic handle seeing two different "versions" of the same data? Thanks to the uniqueness requirement, these questions do not have to be answered.

There are three types of JDO identity, but only two of them are important to most applications: *datastore identity* and *application identity*. The majority of JDO implementations support datastore identity at a minimum; many support application identity as well. All persistent classes in an inheritance tree must use the same form of JDO identity.

Note

Kodo supports both datastore and application identity.

4.5.1. Datastore Identity

Datastore identity is managed by the JDO implementation. It is independent of the values of your persistent fields. You have no control over what class is used for JDO identity objects, and limited control over what data is used to create identity values. The only requirement placed on JDO vendors implementing datastore identity is that the class they use for JDO identity objects meets the following criteria:

- The class must be public.
- The class must be serializable.
- All non-static fields of the class must be public and serializable.
- The class must have a public no-args constructor.
- The class must implement the `toString` method such that passing the result to `PersistenceManager.newObjectIdInstance` creates a new JDO identity object that compares equal to the instance the string was obtained from.

The last criterion is particularly important. As you will see in the chapter on `PersistenceManagers`, it allows you to store the identity of a persistent instance as a string, then later recreate the identity object and retrieve the corresponding persistent instance.

4.5.2. Application Identity

Application identity is managed by you, the developer. Under application identity, the values of one or more persistent fields in an object determine its JDO identity. These fields are called *primary key* fields. Each object's primary key field values must be unique among all other objects of the same type.

When using application identity, the `equals` and `hashCode` methods of the persistence-capable class must depend on all of the primary key fields.

Note

Kodo does not depend upon this behavior. However, some JDO implementations do, so you should implement it if portability is a concern.

If your class has only one primary key field, you can use *single field identity* to simplify working with application identity (see [Section 4.5.3, “Single Field Identity” \[190\]](#)). Otherwise, you must supply an application identity class to use for JDO identity objects. Your application identity class must meet all of the criteria listed for datastore identity classes. It must also obey the following requirements:

- The class must have a string constructor, or a constructor that takes a `Class`, `String` argument pair. The optional `Class` argument is the target persistent class, and the string is the result of `toString` on another identity instance of the same type.
- The names of the non-static fields of the class must include the names of the primary key fields of the corresponding persistent class, and the field types must be identical.
- The `equals` and `hashCode` methods of the class must use the values of all fields corresponding to primary key fields in the persistent class.
- If the class is an inner class, it must be `static`.
- All classes related by inheritance must use the same application identity class, or else each class must have its own application identity class whose inheritance hierarchy mirrors the inheritance hierarchy of the owning persistent classes (see [Section 4.5.2.1, “Application Identity Hierarchies” \[190\]](#))
- Primary key fields must be primitives, primitive wrappers, `Strings`, or `Dates`. Notably, other persistent instances can *not* be used as primary key fields.

Note

For legacy schemas with binary primary key columns, Kodo also supports using primary key fields of type `byte[]`.

These criteria allow you to construct an application identity object from either the values of the primary key fields of a persistent instance, or from a string produced by the `toString` method of another identity object.

Though it is not a requirement, you should also use your application identity class to register the corresponding persistent class with the JVM. This is typically accomplished with a static block in the application identity class code, as the example below illustrates. This registration process is a workaround for a quirk in JDO's persistent type registration system whereby some by-id look-ups might fail if the type being looked up hasn't been used yet in your application.

Note

Though you may still create application identity classes by hand, Kodo provides the `appidtool` to automatically generate proper application identity classes based on your primary key fields. See [Section 5.3.2, “Application Identity Tool” \[456\]](#) of the Reference Guide.

Example 4.5. Application Identity Class

```
/**
 * Persistent class using application identity.
 */
public class Magazine
{
    private String isbn;    // primary key field
    private String title; // primary key field

    /**
     * Equality must be implemented in terms of primary key field
     * equality, and must use instanceof rather than comparing
     * classes directly.
     */
    public boolean equals (Object other)
    {
        if (other == this)
            return true;
        if (!(other instanceof Magazine))
            return false;

        Magazine mag = (Magazine) other;
        return (isbn == mag.isbn
            || (isbn != null && isbn.equals (mag.isbn)))
            && (title == mag.title);
    }
}
```

```

    } || (title != null && title.equals (mag.title));
}
/**
 * Hashcode must also depend on primary key values.
 */
public int hashCode ()
{
    return ((isbn == null) ? 0 : isbn.hashCode ())
        ^ ((title == null) ? 0 : title.hashCode ());
}

// rest of fields and methods omitted

/**
 * Application identity class for Magazine.
 */
public static class MagazineId
{
    static
    {
        // register Magazine with the JVM
        try { Class.forName ("Magazine") } catch (Exception e) {}
    }

    // each primary key field in the Magazine class must have a
    // corresponding public field in the identity class
    public String isbn;
    public String title;

    /**
     * Default constructor requirement.
     */
    public MagazineId ()
    {
    }

    /**
     * String constructor requirement.
     */
    public MagazineId (String str)
    {
        int idx = str.indexOf (':');
        isbn = str.substring (0, idx);
        title = str.substring (idx + 1);
    }

    /**
     * toString must return a string parsable by the string constructor.
     */
    public String toString ()
    {
        return isbn + ":" + title;
    }

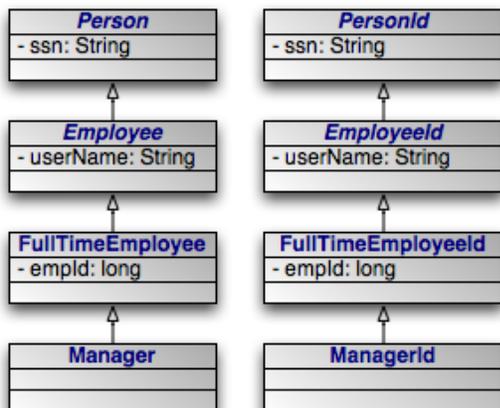
    /**
     * Equality must be implemented in terms of primary key field
     * equality, and must use instanceof rather than comparing
     * classes directly (some JDO implementations may subclass JDO
     * identity class).
     */
    public boolean equals (Object other)
    {
        if (other == this)
            return true;
        if (!(other instanceof MagazineId))
            return false;

        MagazineId mi = (MagazineId) other;
        return (isbn == mi.isbn
            || (isbn != null && isbn.equals (mi.isbn)))
            && (title == mi.title
            || (title != null && title.equals (mi.title)));
    }

    /**
     * Hashcode must also depend on primary key values.
     */
    public int hashCode ()
    {
        return ((isbn == null) ? 0 : isbn.hashCode ())
            ^ ((title == null) ? 0 : title.hashCode ());
    }
}
}

```

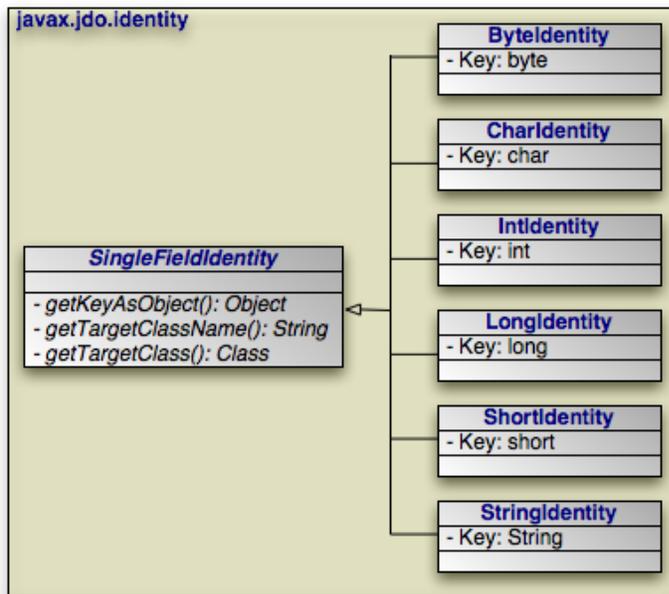
4.5.2.1. Application Identity Hierarchies



An alternative to having a single application identity class for an entire inheritance hierarchy is to have one application identity class per level in the inheritance hierarchy. The requirements for using a hierarchy of application identity classes are as follows:

- The inheritance hierarchy of application identity classes must exactly mirror the hierarchy of the persistent classes that they identify. In the example pictured above, abstract class `Person` is extended by abstract class `Employee`, which is extended by non-abstract class `FullTimeEmployee`, which is extended by non-abstract class `Manager`. The corresponding identity classes, then, are an abstract `PersonId` class, extended by an abstract `EmployeeId` class, extended by a non-abstract `FullTimeEmployeeId` class, extended by a non-abstract `ManagerId` class.
- Subclasses in the application identity hierarchy may define additional primary key fields until the hierarchy becomes non-abstract. In the aforementioned example, `Person` defines a primary key field `ssn`, `Employee` defines additional primary key field `userName`, and `FullTimeEmployee` adds a final primary key field, `empId`. However, `Manager` may not define any additional primary key fields, since it is a subclass of a non-abstract class. The hierarchy of identity classes, of course, must match the primary key field definitions of the persistent class hierarchy.
- It is not necessary for each abstract class to declare primary key fields. In the previous example, the abstract `Person` and `Employee` classes could declare no primary key fields, and the first concrete subclass `FullTimeEmployee` could define one or more primary key fields.
- All subclasses of a concrete identity class must be `equals` and `hashCode`-compatible with the concrete superclass. This means that in our example, a `ManagerId` instance and a `FullTimeEmployeeId` instance with the same primary key field values should have the same hash code, and should compare equal to each other using the `equals` method of either one. In practice, this requirement reduces to the following coding practices:
 1. Use `instanceof` instead of comparing `Class` objects in the `equals` methods of your identity classes.
 2. An identity class that extends another non-abstract identity class should not override `equals` or `hashCode`.

4.5.3. Single Field Identity



Single field identity is a subset of application identity. When you have only one primary key field, you can choose to use one of JDO's built-in single field identity classes instead of coding your own application identity class. All single field identity classes extend `javax.jdo.identity.SingleFieldIdentity`. This base type defines the following methods:

```
public Object getKeyAsObject ()
```

Returns the primary key value as an object. Each `SingleFieldIdentity` subclass also defines a `getKey` method to return the primary key in its primitive form.

```
public Class getTargetClass ()
```

The target class of a single field identity object is the persistent class to which the identity object corresponds. Note that the target class is not part of the serialized state of a `SingleFieldIdentity` instance. After an instance has been deserialized, calls to this method return null.

```
public String getTargetClassName ()
```

Returns the name of the target class. This method returns the correct value even after a single field identity object has been deserialized.

The following list enumerates the primary key field types supported by single field identity, and the built-in identity class for each type:

- byte, `java.lang.Byte`: `javax.jdo.identity.ByteIdentity`

- char, java.lang.Character: **javax.jdo.identity.CharIdentity**
- int, java.lang.Integer: **javax.jdo.identity.IntIdentity**
- long, java.lang.Long: **javax.jdo.identity.LongIdentity**
- short, java.lang.Short: **javax.jdo.identity.ShortIdentity**
- java.lang.String: **javax.jdo.identity.StringIdentity**

4.6. Conclusions

This chapter covered everything you need to know to write persistent class definitions in JDO. JDO implementations cannot use your persistent classes, however, until you complete one additional step: you must create the JDO metadata. The next chapter explores metadata in detail.

Chapter 5. Metadata

JDO requires that you accompany each persistent class with JDO metadata. This metadata serves three primary purposes:

1. To identify persistent classes.
2. To override default JDO behavior.
3. To provide the JDO implementation with information that it cannot glean from simply reflecting on the persistent class.

Metadata is specified as in the eXtensible Markup Language (XML). The Document Type Definition (DTD) for the persistence portion of metadata documents is given in the next section. JDO also standardizes relational mapping metadata and named query metadata, which we discuss in [Chapter 15, Mapping Metadata \[260\]](#) and [Section 11.10, “Named Queries” \[250\]](#) respectively. Do not worry about digesting the entire DTD immediately; we will fully cover each aspect of persistence metadata in turn.

5.1. Persistence Metadata DTD

```
<!ELEMENT jdo (extension*, package+, extension*)>
<!ELEMENT package (extension*, class+, extension*)>
<!ATTLIST package name CDATA ''>

<!ELEMENT class (extension*, field*, fetch-group*, extension*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class persistence-modifier (persistence-capable|persistence-aware)
'persistence-capable'>
<!ATTLIST class identity-type (datastore|application|non durable) #IMPLIED>
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) #IMPLIED>
<!ATTLIST class embedded-only (true|false) #IMPLIED>
<!ATTLIST class detachable (true|false) #IMPLIED>

<!ELEMENT field (extension*, (array|collection|map)?, extension*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier (none|persistent|transactional) #IMPLIED>
<!ATTLIST field default-fetch-group (true|false) #IMPLIED>
<!ATTLIST field null-value (default|exception|none) #IMPLIED>
<!ATTLIST field dependent (true|false) #IMPLIED>
<!ATTLIST field embedded (true|false) #IMPLIED>
<!ATTLIST field primary-key (true|false) 'false'>
<!ATTLIST field fetch-depth CDATA #IMPLIED>

<!ELEMENT array (extension*)>
<!ATTLIST array embedded-element (true|false) #IMPLIED>
<!ATTLIST array dependent-element (true|false) #IMPLIED>

<!ELEMENT collection (extension*)>
<!ATTLIST collection element-type CDATA #IMPLIED>
<!ATTLIST collection embedded-element (true|false) #IMPLIED>
<!ATTLIST collection dependent-element (true|false) #IMPLIED>

<!ELEMENT map (extension*)>
<!ATTLIST map key-type CDATA #IMPLIED>
<!ATTLIST map embedded-key (true|false) #IMPLIED>
<!ATTLIST map dependent-key (true|false) #IMPLIED>
<!ATTLIST map value-type CDATA #IMPLIED>
<!ATTLIST map embedded-value (true|false) #IMPLIED>
<!ATTLIST map dependent-value (true|false) #IMPLIED>

<!ELEMENT fetch-group (field)*>
<!ATTLIST fetch-group name CDATA #REQUIRED>

<!ELEMENT extension ANY>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

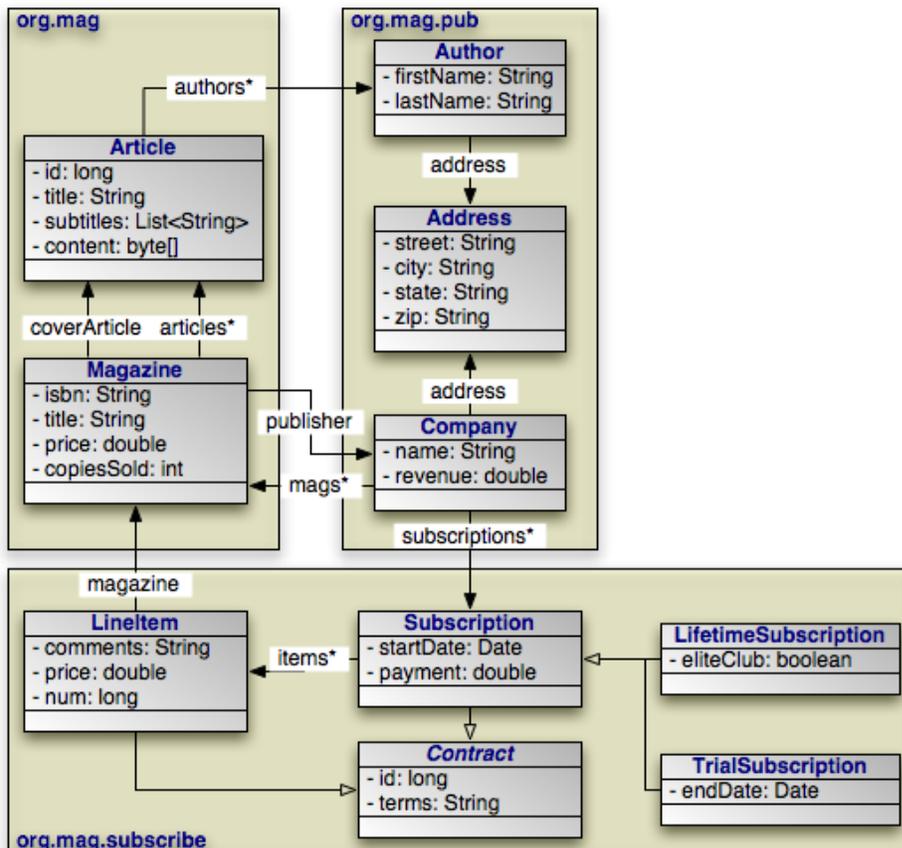
5.2. JDO, Package, and Extension Elements

The root element of all persistence metadata documents is the `jdo` element. The only legal children of the `jdo` element are `package` elements and `extension` elements. Each `package` element must specify a `name` attribute giving the full name of the package it represents. Extensions are used to annotate metadata with vendor-specific information. The `extension` element may contain arbitrary XML content, and has three attributes:

- `vendor-name`: The name of the vendor the extension applies to. This attribute is required.
- `key`: The name of the property you are setting with the extension. Each vendor will supply a list of supported properties.
- `value`: The value of the property.

Note

Kodo defines many useful metadata extensions. See [Section 6.4, “Metadata Extensions” \[483\]](#) in the Reference Guide for the complete set.



Through the course of this chapter, we will build a persistence metadata document for the model above. Our first step is to define the basic metadata structure using the elements we have examined so far:

Example 5.1. Basic Structure of Metadata Documents

```
<?xml version="1.0"?>
<jdo>
  <package name="org.mag">
    . . .
  </package>
  <package name="org.mag.pub">
    . . .
  </package>
  <package name="org.mag.subscribe">
    . . .
  </package>
</jdo>
```

5.3. Class Element

package elements contain one or more class elements, surrounded by zero or more extension elements. Every persistent class in the package named by each package element must be represented by a class element. Before we explore this element in detail, a brief note on how JDO resolves class names is in order.

Several metadata attributes require you to specify class names. The names you give should follow these guidelines:

- If the class is in the package named by the current package element, you can give just the class name, without specifying the package. For example, if the current package name is `org.mag` and the class is `org.mag.Magazine`, then you can simply write `Magazine` for the class name.
- Similarly, if the class is in `java.lang`, `java.util`, or `java.math` packages, you do not need to specify the package in the class name.
- Otherwise, use the full class name, including package name.
- If the class is an inner class, then write it as `<parent-class>${<inner-class>}`. For example, `Subscription$LineItem`.

We now turn our attention back to the class element. This element has the following attributes:

- `name`: The unqualified name of the class. This attribute is required.
- `persistence-modifier`: This attribute defaults to `persistence-capable`, denoting a persistent class. You can, however, also list persistence-aware classes in metadata by setting this attribute to `persistence-aware` (see [Section 4.2, “Persistence-Capable vs. Persistence-Aware” \[180\]](#) for a discussion of persistence-aware classes). In this case, the class metadata should have no other content. Listing persistence-aware classes in metadata is optional.
- `identity-type`: Gives the JDO identity type used by the class. Legal values are `application` for application identity, `datastore` for datastore identity, and `none`. This attribute defaults to a value of `application` if you declare any primary key fields or an `objectid-class`, and `datastore` otherwise.
- `objectid-class`: For application identity, the name of the JDO identity class used by this persistent class. If this is a persistent subclass that uses the same `objectid-class` as its superclass, you do not have to specify this attribute. Do not specify this attribute if you use single field identity (see [Section 4.5.3, “Single Field Identity” \[190\]](#)).
- `requires-extent`: Set this attribute to `false` if you will never need to query for persistent instances of this class (i.e., if all objects of the class can be obtained through JDO identity lookups or through relations with other objects). Defaults to `true`.
- `embedded-only`: Set this attribute to `true` to indicate that this class will only be used in embedded relations, and does not

require any independent representation in the datastore. When `embedded-only` is true, `requires-extent` defaults to false.

- `detachable`: Whether instances of this class can be detached from a `PersistenceManager` and later re-attached to apply offline changes. The default is false. See [Section 8.7, “Detach and Attach Functionality” \[219\]](#) for details on detachment.

Example 5.2. Metadata Class Listings

```
<?xml version="1.0"?>
<jdo>
  <package name="org.mag">
    <!-- application identity -->
    <class name="Magazine" objectid-class="Magazine$MagazineId">
      ...
    </class>
    <!-- single field identity -->
    <class name="Article" identity-type="application">
      ...
    </class>
  </package>
  <package name="org.mag.pub">
    <!-- default datastore identity -->
    <class name="Company">
      ...
    </class>
    <class name="Author">
      ...
    </class>
    <class name="Address" embedded-only="true">
      ...
    </class>
  </package>
  <package name="org.mag.subscribe">
    <!-- single field identity -->
    <class name="Contract" identity-type="application">
      ...
    </class>
    <class name="Subscription">
      ...
    </class>
    <class name="LifetimeSubscription">
      ...
    </class>
    <class name="TrialSubscription">
      ...
    </class>
    <!-- static inner class -->
    <class name="Subscription$LineItem">
      ...
    </class>
  </package>
</jdo>
```

5.4. Field Element

The class element may contain extension elements, field elements, and `fetch-group` elements. field elements represent fields declared by the persistent class. These elements are optional; if a field declared in the class is not named by some field element, then its properties are defaulted as explained in the attribute listings below. Thanks to JDO's comprehensive set of defaults, most fields do not need to be listed explicitly. field elements accept the following attributes:

- `name`: The name of the field, as it is declared in the persistent class. This attribute is required.
- `persistence-modifier`: Specifies how JDO should manage the field. Legal values are `persistent` for persistent fields, `transactional` for fields that are non-persistent but can be rolled back along with the current **transaction**, and `none`. The default value of this attribute is based on the type of the field:

- Fields declared `static`, `transient`, or `final` default to `none`.
- Fields of any primitive or primitive wrapper type default to `persistent`.
- Fields of types `java.lang.String`, `java.lang.Number`, `java.math.BigDecimal`, `java.math.BigInteger`, `java.util.Locale`, and `java.util.Date` default to `persistent`.
- Fields of any user-defined persistence-capable type default to `persistent`.
- Arrays of any of the types above default to `persistent`.
- Fields of the following container types in the `java.util` package default to `persistent`: `Collection`, `Set`, `List`, `Map`, `ArrayList`, `HashMap`, `HashSet`, `Hashtable`, `LinkedList`, `TreeMap`, `TreeSet`, `Vector`.
- All other fields default to `none`.
- `primary-key`: Set this attribute to `true` if the class uses application identity and this field is a primary key field. Defaults to `false`.
- `null-value`: Specifies the treatment of `null` values when the field is written to the datastore. Use a value of `none` if the data store should hold a `null` value for the field. Use `default` to write a datastore default value instead. Finally, use `exception` if you want the JDO implementation to throw an exception if the field is `null` at flush time. Defaults to `none`.
- `default-fetch-group`: Default fetch group fields are managed as a single unit for efficiency. They are typically loaded as a block from the datastore, and are often written as a block as well. This attribute defaults to `true` for primitive, primitive wrapper, `String`, `Date`, `BigDecimal`, and `BigInteger` types. All other types default to `false`.
- `dependent`: Set this attribute to `true` to create a *dependent* relation. In a dependent relation, the referenced object is deleted whenever the owning object is deleted, or whenever the relation is severed by nulling or resetting the owning field. For example, if the `Magazine.coverArticle` field is marked `dependent`, then setting `Magazine.coverArticle` to a new `Article` instance will automatically delete the old `Article` stored in the field. Similarly, deleting a `Magazine` object will automatically delete its current `cover Article`.

The `dependent` attribute is only meaningful if the field holds a reference to another persistence-capable object.

Note

Kodo allows you to assign a dependent object that has been severed from its owning field or whose owner has been deleted to another relation within the same transaction. This will prevent the dependent object from being deleted.

- `embedded`: This is a hint to the JDO implementation to store the field as part of the class instance in the datastore, rather than as a separate entity. For example, a relational implementation might store a `Company`'s `Address` properties in the same database row as the `Company` record. `Embedded` defaults to `true` for primitive, primitive wrapper, `Date`, `BigDecimal`, `BigInteger`, array, collection, and map types. All other types default to `false`. Embedded objects do not appear in the `Extent` for their class, and cannot be retrieved directly by query.

Any embedded relation to another persistence-capable object is automatically *dependent* as well.

All field elements may contain extension child elements. field elements that represent array, collection, or map fields may also contain a single array, collection, or map child element, respectively. Each of these elements may contain additional extension elements in turn.

The array element has two attributes:

- `dependent-element`: Equivalent to the field element's `dependent` attribute, but applies to the values stored in each

array index. In addition to the owning field being reset or nulled, dependent elements can be severed from their owner by removing the element from the array. This will cause the element value to be deleted.

- `embedded-element`: This attribute mirrors the `embedded` attribute of the `field` element, but applies to the values stored in each array index.

The `collection` element also has the `dependent-element` and `embedded-element` attributes. Additionally, it declares the `element-type` attribute. Use this attribute to tell the JDO implementation what class of objects the collection contains. This is important for efficient storage of the collection contents. The `element-type` defaults to `java.lang.Object` unless you are using Java 5 parameterized types, in which case it defaults to the type parameter in your field declaration.

The `map` element defines six attributes. They are:

- `key-type`: The class of objects used for map keys. Similar to `element-type` as described above. The `key-type` defaults to `java.lang.Object` unless you are using Java 5 parameterized types, in which case it defaults to the key type parameter in your field declaration.
- `dependent-key`: Same as the `dependent-element` attribute of arrays and collections, but applies to map keys.
- `embedded-key`: Same as the `embedded-element` attribute of arrays and collections, but applies to map keys.
- `value-type`: The class of objects used for map values. The `value-type` defaults to `java.lang.Object` unless you are using Java 5 parameterized types, in which case it defaults to the value type parameter in your field declaration.
- `dependent-value`: Same as the `dependent-element` element of arrays and collections, but applies to map values.
- `embedded-value`: Same as the `embedded-element` element of arrays and collections, but applies to map values.

5.5. Fetch Group Element

Fetch groups are sets of fields that are loaded together. You may recall that a field can use the `default-fetch-group` attribute to control whether it is in the default fetch group. JDO also allows you to create other fetch groups. As you will see in [Chapter 12, *FetchPlan* \[254\]](#), you can use fetch groups in conjunction with JDO's `FetchPlan` interface to fine-tune data loading.

You create fetch groups with the `fetch-group` metadata element. This element goes within the `class` element, after all `field` elements. Classes can define multiple fetch groups. The `fetch-group` element has the following attributes:

- `name`: The name of the fetch group. Fetch group names are global, and are expected to be shared among classes. For example, a shopping website may use a *detail* fetch group in each product class to efficiently load all the data needed to display a product's "detail" page. The website might also define a sparse *list* fetch group containing only the fields needed to display a table of products, as in a search result.

The following names are reserved for use by JDO: `default`, `values`, `all`, `none`, and any name beginning with `jdo`.

Note

Kodo also reserves fetch group names beginning with `ejb` or `kodo`.

Each `fetch-group` contains `field` elements. As you might expect, listing a `field` within a `fetch-group` includes that field in the fetch group. Each fetch group `field` can have the following attributes:

- `name`: The name of the persistent field.
- `fetch-depth`: If the field represents a relation, the depth to which to recurse. The current fetch group will be applied to the related object when fetching it, and so on until the depth is exhausted or the related object has no relation fields in the current fetch group. Under the default depth of 1, the related object will be fetched, but none of its relation fields will be traversed, even if they are in the current fetch group. With a depth of 2, the related object will be fetched, and if it has any relation fields in the current fetch group, those will be fetched with a depth of 1. A depth of 0 indicates that the recursion continues until the graph is exhausted or a related object has no relation fields in the current fetch group.

Thus, to create a *detail* fetch group consisting of the `publisher` and `articles` relations, with the fetch group applied recursively to the related objects, use:

```
<fetch-group name="detail">
  <field name="publisher" fetch-depth="0"/>
  <field name="articles" fetch-depth="0"/>
</fetch-group>
```

Note

Kodo currently places the following restrictions on fetch groups:

1. A given field may be in only one fetch group, including the default fetch group.
2. All relation fields included in a custom fetch group must set their `fetch-depth` to 0, meaning the fetch group will be applied recursively to arbitrary depth.

The behavior mandated by these restrictions mimics the behavior of previous Kodo versions.

5.6. The Complete Document

That exhausts the metadata document structure. We present the complete metadata document for our sample model below:

Example 5.3. Full Metadata Document

```
<?xml version="1.0"?>
<!-- Note that all persistence-capable classes must be listed, but -->
<!-- very few fields need to be specified -->
<jdo>
  <package name="org.mag">
    <class name="Magazine" objectid-class="Magazine$MagazineId">
      <field name="isbn" primary-key="true"/>
      <field name="title" primary-key="true"/>
      <field name="articles">
        <collection element-type="Article" dependent-element="true"/>
      </field>
      <fetch-group name="detail">
        <field name="publisher" fetch-depth="0"/>
        <field name="articles" fetch-depth="0"/>
      </fetch-group>
    </class>
    <class name="Article" identity-type="application" detachable="true">
      <field name="id" primary-key="true"/>
      <field name="authors">
        <map key-type="String" value-type="org.mag.pub.Author"/>
      </field>
      <field name="subtitles">
        <collection element-type="String"/>
      </field>
    </class>
  </package>
</jdo>
```

```
</class>
</package>
<package name="org.mag.pub">
  <class name="Company">
    <field name="address" embedded="true"/>
    <field name="subscriptions">
      <collection element-type="org.mag.subscribe.Subscription"/>
    </field>
    <field name="mags">
      <collection element-type="org.mag.Magazine"/>
    </field>
  </class>
  <class name="Author">
    <field name="address" embedded="true"/>
  </class>
  <class name="Address" embedded-only="true"/>
</package>
<package name="org.mag.subscribe">
  <class name="Contract" identity-type="application">
    <field name="id" primary-key="true"/>
  </class>
  <class name="Subscription" detachable="true">
    <field name="items">
      <collection element-type="Subscription$LineItem" dependent-element="true"/>
      <extension vendor-name="kodo" key="lock-group" value="none"/>
    </field>
  </class>
  <class name="LifetimeSubscription"/>
  <class name="TrialSubscription"/>
  <class name="Subscription$LineItem"/>
</package>
</jdo>
```

5.7. Metadata Placement

JDO metadata must be available both during class enhancement and at runtime. The metadata document listing a persistent class must be available as a resource from the class' class loader, and must exist in one of two standard locations:

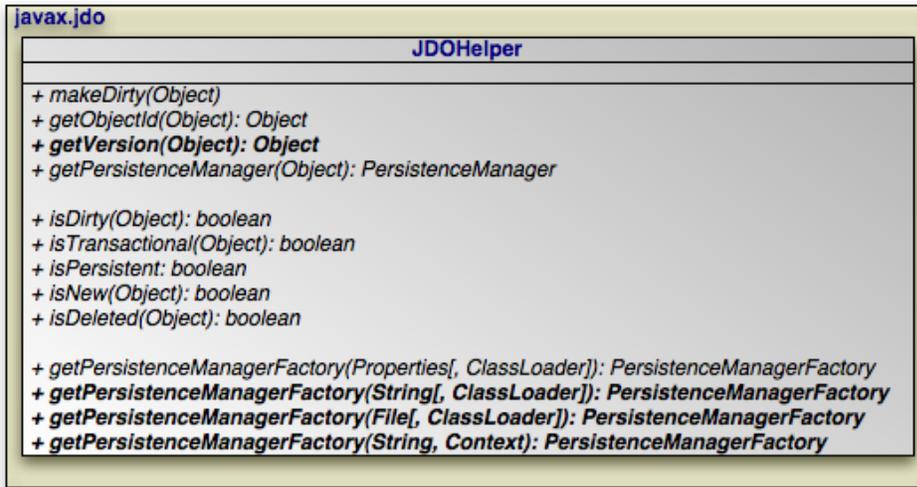
1. In a resource called `<class-name>.jdo`, where `<class-name>` is the name of the class the document applies to, without package name. The resource must be located in the same package as the class.
2. In a resource called `package.jdo`. The resource should be placed in the corresponding package, or in any ancestor package. Package-level documents contain the metadata for all the persistence-capable classes in the package, except those classes that have individual `class-name.jdo` resources. Package-level documents may also contain the metadata for classes in any sub-packages.

Assuming you are using a standard Java class loader, these rules imply that for a class `Magazine` defined by the file `org/mag/Magazine.class`, you can define the corresponding metadata in any of the following files:

- `org/mag/Magazine.jdo`
- `org/mag/package.jdo`
- `org/package.jdo`
- `package.jdo`

Because metadata documents are loaded as resources, JDO implementations can also read them from `jar` files.

Chapter 6. JDOHelper



Note

Kodo includes the `KodoJDOHelper` helper class to provide additional utility methods.

Applications use the `JDOHelper` for three types of operations: persistence-capable operations, lifecycle operations, and `PersistenceManagerFactory` construction. We investigate each below.

6.1. Persistence-Capable Operations

```
public static void makeDirty (Object pc, String fieldName);
public static Object getObjectId (Object pc);
public static Object getVersion (Object pc);
public static PersistenceManager getPersistenceManager (Object pc);
```

We have already seen the first two persistence-capable operations, `makeDirty` and `getObjectId`. Given a persistence-capable object and the name of the field that has been modified, the `makeDirty` method notifies the JDO implementation that the field's value has changed so that it can write the new value to the datastore. JDO usually tracks field modifications automatically; the only time you are required to use this method is when you assign a new value to some index of a persistent array.

The `getObjectId` method returns the JDO identity object for the persistence-capable instance given as an argument. If the given instance is not persistent, this method returns `null`.

Note

Under some settings, your JDO implementation may have to insert a newly-persisted object into the datastore before it can determine the object's JDO identity. Thus, calling `JDOHelper.getObjectId` on an object that has been made persistent in the current transaction might cause the `PersistenceManager` to *flush*. Flushing is described in [Chapter 8, `PersistenceManager` \[214\]](#)

The `getVersion` method returns the version object of the given persistence-capable instance. Version objects are used to en-

sure that concurrent optimistic transactions don't overwrite each other's changes. For more information on optimistic transactions, see [Section 9.1, “Transaction Types” \[226\]](#) Standard JDO versioning strategies are discussed in [Section 15.10, “Version” \[284\]](#)

If the argument to `getVersion` is not persistent or does not have a version, the method returns `null`.

The final persistence-capable operation, `getPersistenceManager`, is self-explanatory. It simply returns the `PersistenceManager` that is managing the persistence-capable object supplied as an argument. If the argument is a *transient* object, meaning it is not managed by a `PersistenceManager`, `null` is returned.

6.2. Lifecycle Operations

```
public static boolean isDirty (Object pc);
public static boolean isTransactional (Object pc);
public static boolean isPersistent (Object pc);
public static boolean isNew (Object pc);
public static boolean isDeleted (Object pc);
```

JDO recognizes several lifecycle states for persistence-capable objects. Instances transition between these states according to strict rules defined in the JDO specification. State transitions can be triggered by both explicit actions, such as calling the `deletePersistent` method of a `PersistenceManager` to delete a persistent object, and by implicit actions, such as reading or writing a persistent field.

The list below enumerates the lifecycle states for persistence-capable instances. Unless otherwise noted, each state must be supported by all JDO implementations. Do not concern yourself with memorizing the states and transitions presented; you will rarely need to think about them in practice.

Note

Some of the state transitions below occur at transaction boundaries. If you are unfamiliar with transactions, you may want to read the first few paragraphs of [Chapter 9, Transaction \[226\]](#) become familiar with the concepts involved before continuing.

- *Transient*. Objects that are created via a user-defined constructor and have no association with the persistence framework are called transient objects. Transient objects behave exactly as if JDO does not exist.
- *Persistent-new*. The persistent-new state is reserved for objects that have been made persistent within the current transaction. On flush or transaction commit, the information in the persistent-new object is inserted into the datastore. On transaction rollback, a persistent-new instance returns to the transient state. The datastore is not affected. If the `Transaction's RestoreValues` property is set to `true`, the instance's persistent and transactional fields will be restored to the values they had when the transaction began.
- *Persistent-new-deleted*. Objects that have been both persisted and then deleted in the current transaction wind up in the persistent-new-deleted state. When objects are in this state, you are only allowed to access their primary key fields. Attempting to access any other persistent field will result in a `JDOUserException`.

A persistent-new-deleted object transitions to transient on transaction commit. The values of its persistent fields are replaced with Java default values. A persistent-new-deleted object also becomes transient if the transaction is rolled back. In this case, its persistent and transactional fields will be restored to the values they had when the transaction began if the `Transaction's RestoreValues` property is `true`, else they will be left untouched.

- *Persistent-clean*. Objects that represent specific state in the datastore and whose persistent fields have not been changed in the current transaction are persistent-clean.
- *Persistent-dirty*. Persistent objects that have been changed within the current transaction are persistent-dirty. On transaction commit, the datastore will be updated to reflect the object's persistent state.

- *Persistent-deleted.* If a persistent object is the parameter of a call to the `PersistenceManager.deletePersistent` method, is deleted by query, or is orphaned from a dependent relation, it becomes persistent-deleted. When an object is in this state, you are only allowed to access its primary key fields. Attempting to access any other persistent field may result in a `JDOUserException`.

A persistent-deleted object transitions to transient on transaction commit. The datastore record for the object is removed.

- *Hollow.* Persistent objects whose values have not been loaded from the datastore are in the hollow state. Whenever an instance transitions to hollow, its persistent fields are cleared and replaced with their Java default values. The fields will be reloaded with their datastore values the first time you access them. Delaying the loading of persistent information until it is needed is known as *lazy loading*.

JDO implementations use only weak or soft references to track hollow instances, so they may be garbage collected if your application does not hold strong references to them.

- *Persistent-nontransactional.* Persistent-nontransactional objects represent persistent data in the datastore, but are not guaranteed to reflect the most current values of that data. A lifecycle state that allows access to data that may be stale might sound useless; if they are utilized carefully, however, persistent-nontransactional objects can sometimes offer large performance gains, with little danger of employing outdated data in your application.

The persistent-nontransactional state is an optional feature of JDO, and may not be supported in some implementations. It is also by far the most complex lifecycle state. It is governed by the `NontransactionalRead`, `NontransactionalWrite`, `RetainValues`, and `Optimistic` properties of the `Transaction`. Implementations may support any or all of these properties. These properties are detailed in [Section 7.2.2, “PersistenceManager and Transaction Defaults” \[209\]](#)

Outside of a transaction, reading and writing persistent fields of a persistent-nontransactional instance does not result in any state change. If the instance enters a datastore transaction, its persistent fields are discarded. Within this type of transaction, reading a persistent field of a persistent-nontransactional instance causes a transition to persistent-clean, and writing a persistent field causes a transition to persistent-dirty.

When a persistent-nontransactional instance enters an optimistic transaction, on the other hand, its persistent field values are retained. Within an optimistic transaction, reading a persistent field of a persistent-nontransactional instance does not change the instance's state; writing a persistent field causes a transition to persistent-dirty.

- *Transient-clean.* The transient-clean and transient-dirty states are grouped together in the *transient-transactional* lifecycle category. Transient-transactional objects are not persistent, but their fields can be restored to their previous values when a transaction is rolled back. You can make a transient instance transient-transactional by passing it to the `PersistenceManager`'s `makeTransactional` method. Some JDO vendors may not support the transient-transactional states; they are an optional feature of the JDO specification.
- *Transient-dirty.* Transient-transactional instances that have been modified in the current transaction are transient-dirty. On transaction completion, a transient-dirty object transitions to transient-clean. If the `Transaction` is rolled back and its `RestoreValues` property is `true`, the persistent and transactional fields of a transient-dirty object will be restored to the values they had when the transaction began.

Note

Kodo supports all JDO lifecycle states, including all optional states.

The following diagram displays the state transitions for persistent objects. Each arrow represents a change from one state to another, and the text next to the arrow indicates the event that triggers change. Method names in blue are methods of the `Transaction` interface. Method names in red are methods of the `PersistenceManager` interface. These interfaces are covered later in this document.

	Persistent	Transactional	Dirty	New	Deleted
Persistent-Dirty	X	X	X		
Persistent-Deleted	X	X	X		X
Persistent-New-Deleted	X	X	X	X	X

6.3. PersistenceManagerFactory Construction

```
public static PersistenceManagerFactory getPersistenceManagerFactory (Properties props);
public static PersistenceManagerFactory getPersistenceManagerFactory (Properties props, ClassLoader loader);
public static PersistenceManagerFactory getPersistenceManagerFactory (String rsrc);
public static PersistenceManagerFactory getPersistenceManagerFactory (String rsrc, ClassLoader loader);
public static PersistenceManagerFactory getPersistenceManagerFactory (File file);
public static PersistenceManagerFactory getPersistenceManagerFactory (File file, ClassLoader loader);
```

You can use the `getPersistenceManagerFactory` methods of `JDOHelper` to obtain `PersistenceManagerFactory` objects in a vendor-neutral fashion. Each method accepts a `java.util.Properties` instance, the name of a CLASSPATH properties resource, or a properties file. `JDOHelper` uses the `javax.jdo.PersistenceManagerFactoryClass` property value to invoke your vendor's `PersistenceManagerFactory` class. It uses the rest of the properties to configure the `PersistenceManagerFactory` before returning it to you. Vendors may construct a new `PersistenceManagerFactory` with each invocation of this method, or may return a pooled instance that matches the supplied properties. The available configuration options and their associated property names are discussed in [Chapter 7, PersistenceManagerFactory \[207\]](#)

Note

Set the `javax.jdo.PersistenceManagerFactoryClass` to `kodo.jdo.PersistenceManagerFactoryImpl` to use Kodo.

If the vendor's `PersistenceManagerFactory` class is not visible to the current thread's class loader, you can supply an alternative `ClassLoader` to the `JDOHelper` when invoking each method.

```
public static PersistenceManagerFactory getPersistenceManagerFactory (String name,
    Context jndiContext);
public static PersistenceManagerFactory getPersistenceManagerFactory (String name,
    Context jndiContext, ClassLoader loader);
```

This is a convenience method to retrieve a previously-bound `PersistenceManagerFactory` from the Java Naming and Directory Interface (JNDI). If the given `Context` is null, the `JDOHelper` will create a new `InitialContext` with which to perform the lookup.

Example 6.1. Obtaining a PersistenceManagerFactory from Properties

```
// this is usually just done once in your application somewhere, and then
// you cache the factory for easy retrieval by application components
Properties props = new Properties ();

// this property key tells the JDOHelper what factory class to instantiate
props.setProperty ("javax.jdo.PersistenceManagerFactoryClass",
    "kodo.jdo.PersistenceManagerFactoryImpl");

// these properties define the default settings for PersistenceManagers
```

```
// produced by this factory; these settings are covered in the next chapter
props.setProperty ("javax.jdo.option.Optimistic", "true");
props.setProperty ("javax.jdo.option.RetainValues", "true");
props.setProperty ("javax.jdo.option.ConnectionUserName", "solarmetric");
props.setProperty ("javax.jdo.option.ConnectionPassword", "kodo");
props.setProperty ("javax.jdo.option.ConnectionURL", "jdbc:hsqldb:database");
props.setProperty ("javax.jdo.option.ConnectionDriverName",
    "org.hsqldb.jdbcDriver");

PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory (props);
```

Example 6.2. Obtaining a PersistenceManagerFactory from a Resource

```
// the jdo.properties resource sets the javax.jdo.PersistenceManagerFactory
// key to kodo.jdo.PersistenceManagerFactoryImpl, and includes other
// javax.jdo.option.* settings
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory ("jdo.properties");
```

Chapter 7. PersistenceManagerFactory

```
javax.jdo
PersistenceManagerFactory
- ConnectionUserName: String
- ConnectionPassword: String
- ConnectionURL: String
- ConnectionDriverName: String
- ConnectionFactoryName: String
- ConnectionFactory2Name: String
- ConnectionFactory: Object
- ConnectionFactory2: Object

- Multithreaded: boolean
- Optimistic: boolean
- RetainValues: boolean
- RestoreValues: boolean
- NontransactionalRead: boolean
- NontransactionalWrite: boolean
- IgnoreCache: boolean
- Mapping: String
- DetachAllOnCommit: boolean
- addInstanceLifecycleListener(InstanceLifecycleListener, Class[])
- removeInstanceLifecycleListener(InstanceLifecycleListener)

- getPersistenceManager(): PersistenceManager
- getPersistenceManager(String, String): PersistenceManager

- supportedOptions(): Collection
- getProperties(): Properties

- getDataStoreCache(): DataStoreCache

- isClosed(): boolean
- close()
```

The `PersistenceManagerFactory` creates `PersistenceManager` instances for application use. It allows you to configure datastore connectivity and to specify the default settings of the `PersistenceManagers` it constructs. You can also use it to programmatically discover what JDO options your current vendor supports, enabling you to build applications that optimize themselves for full-featured products, but still function under more basic JDO implementations.

Note

Kodo extends the standard `PersistenceManagerFactory` with the `KodoPersistenceManagerFactory` interface to provide additional functionality.

7.1. Obtaining a PersistenceManagerFactory

JDO vendors may supply public constructors for their `PersistenceManagerFactory` implementations, but the recommended method of obtaining a `PersistenceManagerFactory` is through the Java Connector Architecture (JCA) in a managed environment, or through the `JDOHelper`'s `getPersistenceManagerFactory` methods in an unmanaged environment, as described in [Section 6.3, “PersistenceManagerFactory Construction”](#) [205]. `PersistenceManagerFactories` obtained through these means are immutable; any attempt to change their property settings will result in a `JDOUserException`. This is because the returned factory may come from a pool, and might be shared by other application components.

JDO requires that concrete `PersistenceManagerFactory` classes implement the `Serializable` interface. This allows you to create and configure a `PersistenceManagerFactory`, then serialize it to a file or store it in a Java Naming and Dir-

actory Interface (JNDI) tree for later retrieval and use.

7.2. PersistenceManagerFactory Properties

The majority of the `PersistenceManagerFactory` interface consists of Java bean-style "getter" and "setter" methods for several properties, represented by field declarations in the diagram at the beginning of this chapter. These properties can be grouped into two functional categories: datastore connection configuration and default `PersistenceManager` and `Transaction` options.

The sections below explain the meaning of each property. Many of these properties can be set through the `Properties` instance supplied to the aforementioned `getPersistenceManagerFactory` methods in `JDOHelper`. Where this is the case, the `Properties` key is displayed along with the method declarations.

7.2.1. Connection Configuration

Use the properties below to tell JDO implementations how to connect with your datastore.

```
public String getConnectionUserName ();
public void setConnectionUserName (String user);
javax.jdo.option.ConnectionUserName
```

The user name to specify when connecting to the datastore.

```
public String getConnectionPassword ();
public void setConnectionPassword (String pass);
javax.jdo.option.ConnectionPassword
```

The password to specify when connecting to the datastore.

```
public String getConnectionURL ();
public void setConnectionURL (String url);
javax.jdo.option.ConnectionURL
```

The URL of the datastore.

```
public String getConnectionDriverName ();
public void setConnectionDriverName (String driver);
javax.jdo.option.ConnectionDriverName
```

The full class name of the driver to use when interacting with the datastore.

```
public String getConnectionFactoryName ();
public void setConnectionFactoryName (String name);
javax.jdo.option.ConnectionFactoryName
```

The JNDI location of a datastore connection factory. This property overrides the other datastore connection properties above.

```
public Object getConnectionFactory ();
public void setConnectionFactory (Object factory);
```

A datastore connection factory. This property overrides all other datastore connection properties above, including the `ConnectionFactoryName`. The exact type of the given factory is implementation-dependent. Many JDO implementations will expect a standard Java Connector Architecture (JCA) `ConnectionFactory`. Implementations layered on top of JDBC might expect a `JDBC DataSource`. Still other implementations might use other factory types.

Note

Kodo uses `JDBC DataSource`s as connection factories.

```
public String getConnectionFactory2Name ();
public void setConnectionFactory2Name (String name);
javax.jdo.option.ConnectionFactory2Name
```

In a managed environment, connections obtained from the primary connection factory may be automatically enlisted in any global transaction in progress. The JDO implementation might require an additional connection factory that is not configured to participate in global transactions. For example, Kodo's default algorithm for datastore identity generation requires its own non-managed transaction. Specify the JNDI location of this factory through this property.

```
public Object getConnectionFactory2 ();
public void setConnectionFactory2 (Object factory);
```

The connection factory used for local transactions. Overrides the `ConnectionFactory2Name` above.

7.2.2. PersistenceManager and Transaction Defaults

The settings below will become the default property values for the `PersistenceManagers` and associated `Transactions` produced by the `PersistenceManagerFactory`. Some implementations may not fully support all properties. If you attempt to set a property to an unsupported value, the operation will throw a `JDOUnsupportedOptionException`.

```
public boolean getMultithreaded ();
public void setMultithreaded (boolean multithreaded);
javax.jdo.option.Multithreaded
```

Set this property to `true` to indicate that `PersistenceManagers` or the persistence-capable objects they manage will be accessed concurrently by multiple threads in your application. Some JDO implementations might optimize performance by avoiding any synchronization when this property is left `false`.

```
public boolean getOptimistic ();
public void setOptimistic (boolean optimistic);
javax.jdo.option.Optimistic
```

Set to `true` to use optimistic transactions by default. **Section 9.1, “Transaction Types” [226]** discusses optimistic transactions.

```
public boolean getRetainValues ();
public void setRetainValues (boolean retain);
javax.jdo.option.RetainValues
```

If this property is `true`, the fields of persistent objects will not be cleared on transaction commit. Otherwise, persistent fields are cleared on commit and re-read from the datastore the next time they are accessed.

```
public boolean getRestoreValues ();
public void setRestoreValues (boolean restore);
javax.jdo.option.RestoreValues
```

Controls the behavior of persistent and transactional fields on transaction rollback. Set this property to `true` to restore the fields to the values they had when the transaction began.

```
public boolean getNontransactionalRead ();
public void setNontransactionalRead (boolean read);
javax.jdo.option.NontransactionalRead
```

Specifies whether you can read persistent state outside of a transaction. If this property is `false`, any attempt to iterate an `Extent`, execute a `Query`, or access non-primary key persistent object fields outside of a transaction will result in a `JDOUserException`.

```
public boolean getNontransactionalWrite ();
public void setNontransactionalWrite (boolean write);
javax.jdo.option.NontransactionalWrite
```

Specifies whether you can write to persistent objects and perform persistence operations outside of a transaction. If this property is `false`, any attempt to modify a persistent object outside of a transaction will result in a `JDOUserException`.

Note that changes made outside of a transaction are discarded if the modified object enters a datastore transaction, but retained if it enters an optimistic transaction. We cover transactions in **Chapter 9, Transaction [226]**

```
public boolean getIgnoreCache ();
public void setIgnoreCache (boolean ignore);
javax.jdo.option.IgnoreCache
```

This property controls whether changes made to persistent instances in the current transaction are considered when evaluating queries. A value of `true` is a hint to the JDO runtime that changes in the current transaction can be ignored; this usually enables the implementation to run the query using the datastore's native query interface. A value of `false`, on the other hand, may force implementations to flush changes to the datastore before running queries, or to run transactional queries in memory, both of which can have a negative impact on performance.

```
public boolean getDetachAllOnCommit ();
public void setDetachAllOnCommit (boolean detach);
javax.jdo.option.DetachAllOnCommit
```

PersistenceManagers can optionally detach their entire object cache when a transaction commits. This setting controls the default for PersistenceManagers produced by this factory. We discuss detachment in [Section 8.7, “Detach and Attach Functionality” \[219\]](#)

```
public String getMapping ();
public void setMapping (String mapping);
javax.jdo.option.Mapping
```

The name of the logical mapping between your persistent classes and the datastore. [Section 15.1, “Mapping Metadata Placement” \[260\]](#) describes how the mapping name is used to locate relational mapping metadata.

```
public void addInstanceLifecycleListener (InstanceLifecycleListener listen,
    Class[] classesOfInterest);
public void removeInstanceLifecycleListener (InstanceLifecycleListener listen);
```

Recall from [Section 4.4.2, “InstanceLifecycleListener” \[185\]](#) that InstanceLifecycleListeners consume events fired by lifecycle changes in persistent objects. The methods above allow you to add and remove listeners that will be passed on to all PersistenceManagers the PersistenceManagerFactory creates in the future. See the same-named methods of the PersistenceManager interface in [Section 8.2, “Configuration Properties” \[215\]](#) for API details.

Note

Kodo supports all of the properties above. It recognizes many additional properties as well; see [Chapter 2, Configuration \[393\]](#) of the Reference Guide for details.

7.3. Obtaining PersistenceManagers

```
public PersistenceManager getPersistenceManager ();
public PersistenceManager getPersistenceManager (String user, String pass);
```

The PersistenceManagerFactory interface includes two getPersistenceManager methods for obtaining PersistenceManager instances. One version takes as parameters the user name and password to use for the PersistenceManager's data store connection. The other version relies on the ConnectionUserName and ConnectionPassword settings of the PersistenceManagerFactory. Both methods may return a newly-constructed PersistenceManager, or may return one from a pool of instances.

If the PersistenceManagerFactory was not already immutable, then it will be after you acquire the first PersistenceManager from it. Any attempt to change the factory's properties will result in a JDOUserException.

7.4. Properties and Supported Options

```
public Properties getProperties ();
public Collection supportedOptions ();
```

In addition to creating `PersistenceManagers`, the `PersistenceManagerFactory` also supplies metadata about the current JDO implementation. The `getProperties` method returns a `Properties` instance containing, at a minimum, the following keys:

- `VendorName`: The name of the JDO vendor.
- `VersionNumber`: The version number string for the product.

The `supportedOptions` method returns a `Collection` of `Strings` enumerating the JDO options supported by the implementation. The following option names are recognized:

- `javax.jdo.option.TransientTransactional`: Support for the transient-clean and transient-dirty lifecycle states.
- `javax.jdo.option.NontransactionalRead`: Support for the `NontransactionalRead` property described in [Section 7.2.2, “PersistenceManager and Transaction Defaults” \[209\]](#).
- `javax.jdo.option.NontransactionalWrite`: Support for the `NontransactionalWrite` property described in [Section 7.2.2, “PersistenceManager and Transaction Defaults” \[209\]](#).
- `javax.jdo.option.RetainValues`: Support for the `RetainValues` property described in [Section 7.2.2, “PersistenceManager and Transaction Defaults” \[209\]](#).
- `javax.jdo.option.Optimistic`: Support for the `Optimistic` property described in [Section 7.2.2, “PersistenceManager and Transaction Defaults” \[209\]](#).
- `javax.jdo.option.ApplicationIdentity`: Support for JDO application identity, as described in [Section 4.5.2, “Application Identity” \[187\]](#).
- `javax.jdo.option.DatastoreIdentity`: Support for JDO datastore identity, as described in [Section 4.5.1, “Datastore Identity” \[187\]](#).
- `javax.jdo.option.NonDurableIdentity`: Support for *non-durable identity*, a seldom-used form of JDO identity not covered in this document.
- `javax.jdo.option.ArrayList`: Support for persistent `java.util.ArrayList` fields.
- `javax.jdo.option.LinkedList`: Support for persistent `java.util.LinkedList` fields.
- `javax.jdo.option.TreeMap`: Support for persistent `java.util.TreeMap` fields.
- `javax.jdo.option.TreeSet`: Support for persistent `java.util.TreeSet` fields.
- `javax.jdo.option.Vector`: Support for persistent `java.util.Vector` fields.
- `javax.jdo.option.List`: Support for persistent `java.util.List` fields.
- `javax.jdo.option.Array`: Support for persistent array fields.
- `javax.jdo.option.NullCollection`: This string will be present if the implementation can differentiate between null and empty collections and maps when querying or loading data from the datastore.
- `javax.jdo.option.ChangeApplicationIdentity`: This string will be present if the implementation allows you to change the primary key fields of persistent objects, effectively changing their identity.
- `javax.jdo.option.GetDataStoreConnection`: Whether the `PersistenceManager`.`getDataStoreConnection` is supported.

- `javax.jdo.option.BinaryCompatibility`: This string will be present if the implementation is *binary compatible* with the JDO reference enhancer.
- `javax.jdo.option.UnconstrainedQueryVariables`: Support for *unconstrained variables* in JDOQL queries. See [Example 11.10, “Unconstrained Variables” \[238\]](#)
- `javax.jdo.query.JDOQL`: Support for JDOQL queries. We discuss queries in [Chapter 11, Query \[232\]](#)
- `javax.jdo.query.SQL`: Support for SQL queries. We discuss SQL queries in [Chapter 17, SQL Queries \[320\]](#)

Vendors may include strings for other options and query languages they support as well.

Note

Kodo currently supports all options except `javax.jdo.option.ChangeApplicationIdentity`, `javax.jdo.option.NonDurableIdentity`, `javax.jdo.option.NullCollection`, and `javax.jdo.option.BinaryCompatibility`.

7.5. DataStoreCache Access

```
public DataStoreCache getDataStoreCache ();
```

The `PersistenceManagerFactory` is the gateway to your vendor's data cache. [Chapter 13, DataStoreCache \[257\]](#) describes JDO's standard `DataStoreCache` API for vendor data caches.

7.6. Closing the PersistenceManagerFactory

```
public boolean isClosed ();  
public void close ();
```

`PersistenceManagerFactories` are heavyweight objects. Each factory might maintain a metadata cache, object state cache, `PersistenceManager` pool, connection pool, and more. If your application no longer needs a `PersistenceManagerFactory`, you should close it to free these resources. When a `PersistenceManagerFactory` closes, all `PersistenceManagers` from that factory, and by extension all persistent objects managed by those `PersistenceManagers`, become invalid. Attempting to close a `PersistenceManagerFactory` while one or more of its `PersistenceManagers` has an active transaction results in a `JDOUserException`.

Closing a `PersistenceManagerFactory` should not be taken lightly. It is much better to keep a factory open for a long period of time than to repeatedly create and close new factories. Thus, most applications will never close the factory, or only close it when the application is exiting. Only applications that require multiple factories with different configurations have an obvious reason to create and close multiple `PersistenceManagerFactory` instances. In fact, because closing a `PersistenceManagerFactory` has such drastic consequences, it requires that the current security context have the `closePersistenceManagerFactory` `JDOPermission`. Once a factory is closed, all methods except `isClosed` throw a `JDOUserException`.

Chapter 8. PersistenceManager

```
javax.jdo
PersistenceManager
- UserObject: Object
- UserObjects: Map
- Multithreaded: boolean
- IgnoreCache: boolean
- DetachAllOnCommit: boolean
- addInstanceLifecycleListener(InstanceLifecycleListener, Class[])
- removeInstanceLifecycleListener(InstanceLifecycleListener)
- currentTransaction(): Transaction
- getFetchPlan(): FetchPlan
- makePersistent(Object): Object
- makePersistentAll(...): ...
- deletePersistent(Object)
- deletePersistentAll(...)
- makeTransient(Object)
- makeTransientAll(...)
- makeTransactional(Object)
- makeTransactionalAll(...)
- makeNonTransactional(Object)
- makeNonTransactionalAll(...)
- refresh(Object)
- refreshAll(...)
- refreshAll(JDOException)
- retrieve(Object)
- retrieveAll(...)
- detachCopy(Object): Object
- detachCopyAll(...): ...
- getObjectById(...): Object
- getObjectsById(...): ...
- getObjectById(Class, Object): Object
- getObjectId(Object): Object
- getObjectIdClass(Class): Class
- newObjectIdInstance(Class, Object): Object
- evict(Object)
- evictAll(...)
- flush()
- checkConsistency()
- getExtent(Class, boolean): Extent
- newQuery(...): Query
- newNamedQuery(String): Query
- getSequence(String): Sequence
- getConnection(): JDOConnection
- isClosed(): boolean
- close()
```

The diagram above presents an overview of the PersistenceManager interface. For a complete treatment of the PersistenceManager API, see the **Javadoc** documentation. Methods whose parameter signatures consist of an ellipsis (...) are overloaded to take multiple parameter types.

Note

Kodo extends the standard `PersistenceManager` interface with the `KodoPersistenceManager` interface to provide additional functionality.

The `PersistenceManager` is the primary interface to the JDO runtime. Each `PersistenceManager` manages a cache of persistent and transactional objects, and has an association with a single `Transaction`.

We divide the methods of the `PersistenceManager` into the following functional categories:

- User object association.
- Configuration properties.
- `Transaction` association.
- `FetchPlan` association.
- Persistence-capable lifecycle management.
- Detach and attach functionality.
- JDO identity management.
- Cache management.
- `Extent`, `Query`, and `Sequence` factory.
- Connection access.
- Closing.

8.1. User Object Association

```
public Object getUserObject ();
public void setUserObject (Object obj);
public Object putUserObject (Object key, Object obj);
public Object getUserObject (Object key);
public Object removeUserObject (Object key);
```

The `PersistenceManager`'s user objects allow you to associate arbitrary objects with each `PersistenceManager`. The given objects are not used in any way by the JDO implementation.

The `getUserObject` and `setUserObject` methods associate a primary user object with the `PersistenceManager`. The `putUserObject`, `getUserObject`, and `removeUserObject` methods, on the other hand, delegate to an internal map of secondary user objects. As in the `java.util.Map` APIs, the `put` and `remove` operations return the object formerly stored under the given key.

8.2. Configuration Properties

```
public boolean getMultithreaded ();
public void setMultithreaded (boolean threaded);
public boolean getIgnoreCache ();
public void setIgnoreCache (boolean ignore);
```

The `PersistenceManager` interface includes "getter" and "setter" methods for two configuration properties: `MultiThreaded` and `IgnoreCache`. These properties are discussed in [Section 7.2.2, “PersistenceManager and Transaction Defaults” \[209\]](#). The `PersistenceManager`'s `IgnoreCache` setting is passed on to all `Query` and `Extent` instances the `PersistenceManager` creates.

```
public void addInstanceLifecycleListener (InstanceLifecycleListener listen,
    Class[] classesOfInterest);
public void removeInstanceLifecycleListener (InstanceLifecycleListener listen);
```

These methods allow you to add and remove listeners that are notified when managed instances change lifecycle states. The lifecycle event framework is discussed in detail in [Section 4.4.2, “InstanceLifecycleListener” \[185\]](#). When adding a new listener, you can supply an array of persistent classes that the listener is interested in. The listener will only receive events for these classes and their subclasses. If the given array is `null`, the listener will receive events for all classes. The types of events the listener receives is based on the `InstanceLifecycleListener` sub-interfaces it implements.

8.3. Transaction Association

```
public Transaction currentTransaction ();
```

Every `PersistenceManager` has a one-to-one relation with a `Transaction` instance. In fact, many vendors use a single class to implement both the `PersistenceManager` and `Transaction` interfaces. If your application requires multiple concurrent transactions, you will use multiple `PersistenceManagers`.

You can retrieve the `Transaction` associated with a `PersistenceManager` through the `currentTransaction` method.

8.4. FetchPlan Association

```
public FetchPlan getFetchPlan ();
```

Each `PersistenceManager` has a reference to a `FetchPlan`. The `FetchPlan` allows you to fine-tune the fetching of persistent state, as [Chapter 12, *FetchPlan* \[254\]](#) describes.

8.5. Persistence-Capable Lifecycle Management

`PersistenceManagers` perform several actions that affect the lifecycle state of persistence-capable instances. Each of these actions is represented by multiple methods in the `PersistenceManager` interface: one method that acts on a single persistence-capable object, such as `makePersistent`, and corresponding methods that accept a collection or array of persistence-capable objects, such as `makePersistentAll`.

```
public Object makePersistent (Object pc);
public Collection makePersistentAll (Collection pcs);
public Object[] makePersistentAll (Object[] pcs);
```

Transitions transient instances to persistent-new, and creates managed copies of detached objects. This action can only be used in the context of an active transaction. On the next flush or commit, the newly persisted instances will be inserted into the datastore, and changes to the detached instances will be flushed to the corresponding datastore records. If you attempt to attach an instance whose representation has changed in the datastore since detachment, the `makePersistent` operation will throw an exception, or the transaction in which you perform the attachment will fail on commit, just as if a normal optimistic conflict were detected. If the attach process fails, the `RollbackOnly` flag is set on the current transaction.

For each argument, `makePersistent[All]` returns the argument instance if it was transient, or a managed copy of the argument instance if it was detached. Thus, when attaching objects, it is important to use the returned copies rather than continuing to act on the detached objects.

```
public void deletePersistent (Object pc);
public void deletePersistentAll (Collection pcs);
public void deletePersistentAll (Object[] pcs);
```

Transitions persistent instances to persistent-deleted, or persistent-new instances to persistent-new-deleted. This action, too, can only be called during an active transaction. The given instances will be deleted from the datastore on the next flush or commit.

```
public void makeTransient (Object pc);
public void makeTransientAll (Collection pcs);
public void makeTransientAll (Object[] pcs);
```

This action transitions persistent instances to transient. The instances immediately lose their association with the `PersistenceManager` and their JDO identity. The datastore records for the instances are not modified.

This action can only be run on clean objects. If it is run on a dirty object, a `JDOUserException` is thrown.

```
public void makeTransactional (Object pc);
public void makeTransactionalAll (Collection pcs);
public void makeTransactionalAll (Object[] pcs);
```

Use this action to make transient instances transient-transactional, or to bring persistent-nontransactional instances into the current transaction. In the latter case, the action must be invoked during an active transaction.

```
public void makeNontransactional (Object pc);
public void makeNontransactionalAll (Collection pcs);
public void makeNontransactionalAll (Object[] pcs);
```

Transitions transient-clean instances to transient, and persistent-clean instances to persistent-nontransactional. Invoking this action on a dirty instance will result in a `JDOUserException`.

```
public void refresh (Object pc);
public void refreshAll (Collection pcs);
public void refreshAll (Object[] pcs);
public void refreshAll ();
public void refreshAll (JDOException je);
```

Use the `refresh` action to make sure the persistent state of an instance is synchronized with the values in the datastore. `refresh` is intended for long-running optimistic transactions in which there is a danger of seeing stale data.

Calling the `refreshAll` method with no parameters acts on all transactional objects in the `PersistenceManager` cache. If there is no transaction in progress, the method is a no-op.

Calling the `refreshAll` method with a `JDOException` refreshes the failed object held by the exception, and recurses on all nested exceptions.

```
public void retrieve (Object pc);
public void retrieveAll (Collection pcs);
public void retrieveAll (Object[] pcs);
public void retrieveAll (Collection pcs, boolean fgOnly);
public void retrieveAll (Object[] pcs, boolean fgOnly);
```

Retrieving a persistent object immediately loads all of the object's persistent fields with their datastore values. You might use this action to make sure an instance's fields are fully loaded before transitioning it to transient. Note, however, that this action is not recursive. That is, if object A has a relation to object B, then passing A to `retrieve` will load B, but will not necessarily fill B's fields with their datastore values.

Setting `fgOnly` to true will ensure that the fields in the current fetch groups are loaded, but may not load other fields.

8.6. Lifecycle Examples

Example 8.1. Persisting Objects

```
// create some objects
Magazine mag = new Magazine ("1B78-YU9L", "JavaWorld");

Company pub = new Company ("Weston House");
pub.setRevenue (1750000D);
mag.setPublisher (pub);
pub.addMagazine (mag);

Article art = new Article ("JDO Rules!", "Transparent Object Persistence");
art.addAuthor (new Author ("Fred", "Hoyle"));
mag.addArticle (art);

// we only need to make the root object persistent; JDO will traverse
// the object graph and make all related objects persistent too
PersistenceManager pm = pmf.getPersistenceManager ();
pm.currentTransaction ().begin ();
pm.makePersistent (mag);
pm.currentTransaction ().commit ();

// or we could continue using the PersistenceManager...
pm.close ();
```

Example 8.2. Updating Objects

```
Magazine.MagazineId mi = new Magazine.MagazineId ();
mi.isbn = "1B78-YU9L";
mi.title = "JavaWorld";

// read a magazine; note that in order to read objects outside of
// transactions you must have the NonTransactionalRead option set
PersistenceManager pm = pmf.getPersistenceManager ();
Magazine mag = (Magazine) pm.getObjectById (mi, true);
```

```

Company pub = mag.getPublisher ();

// updates should always be made within transactions; note that
// there is no code explicitly linking the magazine or company
// with the transaction; JDO automatically tracks all changes
pm.currentTransaction ().begin ();
mag.setPrice (5.99);
pub.setRevenue (1750000D);
pm.currentTransaction ().commit ();

// or we could continue using the PersistenceManager...
pm.close ();

```

Example 8.3. Deleting Objects

```

// assume we have an object id for the company whose subscriptions
// we want to delete
Object oid = ...;

// read a company; note that in order to read objects outside of
// transactions you must have the NonTransactionalRead option set
PersistenceManager pm = pmf.getPersistenceManager ();
Company pub = (Company) pm.getObjectById (oid, true);

// deletes should always be made within transactions
pm.currentTransaction ().begin ();
pm.deletePersistentAll (pub.getSubscriptions ());
pub.getSubscriptions ().clear ();
pm.currentTransaction ().commit ();

// or we could continue using the PersistenceManager...
pm.close ();

```

8.7. Detach and Attach Functionality

A common use case for an application running in a servlet or application server is to "detach" objects from all server resources, modify them, and then "attach" them again. For example, a servlet might store persistent data in a user session between a modification based on a series of web forms. Between each form request, the web container might decide to serialize the session, requiring that the stored persistent state be disassociated from any other resources. Similarly, a client/server application might transfer persistent objects to a client via serialization, allow the client to modify their state, and then have the client return the modified data in order to be saved. This is sometimes referred to as the *data transfer object* or *value object* pattern, and it allows fine-grained manipulation of data objects without incurring the overhead of multiple remote method invocations.

JDO provides support for this pattern using *detach* APIs that allow you to detach a persistent instance. You can then modify the detached instance offline, and attach the instance back into a `PersistenceManager` - either the same one that detached the instance or a new one - using `makePersistent`, which applies the changes to the the datastore.

In order to be able to detach a persistent instance, the metadata for the class must declare that it is eligible for detachment using the `detachable` class attribute. We described this attribute in [Section 5.3, "Class Element" \[195\]](#).

Note

After changing the `detachable` attribute of a class, you must re-enhance it. Detachability requires that the enhancer add additional fields to the class to hold information about the persistent instance's identity and state.

```

public Object detachCopy (Object pc);
public Collection detachCopyAll (Collection pcs);

```

```
public Object[] detachCopyAll (Object[] pcs);
```

Each detach method returns unmanaged copies of the given instances. The copy mechanism is similar to serialization, except that only currently-loaded fields are copied.

When serializing a detachable object, the serialized copy will be in the detached state automatically. You do not need to explicitly detach an object before serializing it.

Whether through `detachCopy` or through serialization, detaching a dirty instance causes the current transaction to flush. This means that when subsequently re-attaching the detached object, JDO assumes that the transaction from which it was originally detached committed. If the transaction rolled back, then either the re-attachment process will throw an exception, or the transaction in which you re-attach the instance will fail on commit, just as if a normal optimistic conflict was detected.

```
public boolean getDetachAllOnCommit ();
public void setDetachAllOnCommit (boolean detach);
```

Set the `DetachAllOnCommit` property to automatically detach the entire `PersistenceManager` cache on commit. Unlike the explicit `detachCopy` methods, the implicit detach-on-commit occurs in-place. Every instance in the `PersistenceManager` cache transitions to the detached state, then the cache is cleared.

Note

Kodo offers many enhancements to JDO detachment functionality, including additional options for automatic detachment. See [Section 11.1, “Detach and Attach” \[577\]](#) in the Reference Guide for details.

Example 8.4. Detaching and Attaching

This example demonstrates a common client/server scenario. The client requests objects and makes changes to them, while the server handles the object lookups and transactions.

```
// CLIENT:
// requests an object with a given oid
Record detached = (Record) getFromServer (oid);

...

// SERVER:
// detaches object and returns to client
Object oid = processClientRequest ();
PersistenceManager pm = pmf.getPersistenceManager ();
Record record = (Record) pm.getObjectById (oid);
// we detach explicitly because we want to close the PM before serializing
Record detached = (Record) pm.detachCopy (record);
pm.close ();
sendToClient (detached);

...

// CLIENT:
// makes some modifications and sends back to server
detached.setSomeField ("bar");
sendToServer (detached);

...

// SERVER:
// re-attaches the instance and commit the changes
Record modified = (Record) processClientRequest ();
PersistenceManager pm = pmf.getPersistenceManager ();
pm.currentTransaction ().begin ();
Record attached = (Record) pm.makePersistent (modified);
```

```
attached.setLastModified (System.currentTimeMillis ());
attached.setModifier (getClientIdentityCode ());
pm.currentTransaction ().commit ();
pm.close ();
```

8.8. JDO Identity Management

Each `PersistenceManager` is responsible for managing the JDO identities of the persistent objects in its cache. The following methods allow you to interact with the management of JDO identities.

```
public Class getObjectIdClass (Class pcClass);
```

Returns the JDO identity class used for the given persistence-capable class. If the given class does not use application identity, returns `null`.

```
public Object newObjectIdInstance (Class pcClass, Object pkOrString);
```

This method is used to re-create JDO identity objects from the string returned by their `toString` method, or, in the case of single field identity, from the primary key value. Given a persistence-capable class and a JDO identity string or primary key object, the method constructs a JDO identity object. Using the `getObjectById` method described below, this identity object can then be employed to obtain the corresponding persistent instance.

Note

Kodo also allows you to pass the primary key value of datastore identity instance into this method to create a datastore identity object.

```
public Object getObjectId (Object pc);
```

Returns the JDO identity object for a persistent instance managed by this `PersistenceManager`. This method is similar to `JDOHelper.getObjectId`.

```
public Object getObjectById (Object oid, boolean validate);
public Object getObjectById (Object oid);
```

These methods returns the persistent instance corresponding to the given JDO identity object. If the instance is already cached, the cached version is returned. Otherwise, a new instance is constructed, and may or may not be loaded with data from the datastore.

If the `validate` parameter of the method is `true`, the JDO implementation will throw a `JDOObjectNotFoundException` if the datastore record for the given JDO identity does not exist. Otherwise, the implementation may return a cached object

without validating it. The implementation might return a hollow instance even when no cached object exists, and an exception will not be thrown until you attempt to access one of the object's persistent fields.

Invoking the method version without a `validate` parameter is equivalent to using a `validate` parameter of `true`.

```
public Object[] getObjectsById (Object[] oids, boolean validate);
public Object[] getObjectsById (Object[] oids);
public Collection getObjectsById (Collection oids, boolean validate);
public Collection getObjectsById (Collection oids);
```

These methods are exactly equivalent to the singular `getObjectById` methods, but acts on an array or collection of identity objects. They may be more efficient than calling the singular method for each identity object.

```
public Object getObjectById (Class pcClass, Object pkOrString);
```

This is a convenience method equivalent to using `newObjectIdInstance` to construct an identity object with the given `Class` and `Object`, then calling `getObjectById` with that identity object and a `validate` parameter of `true`. The code below demonstrates this equivalence.

```
PersistenceManager pm = ...;
Object pkValue = ...;

Object oid = pm.newObjectIdInstance (Magazine.class, pkValue);
Magazine mag1 = (Magazine) pm.getObjectById (oid, true);

Magazine mag2 = (Magazine) pm.getObjectById (Magazine.class, pkValue);
assertTrue (mag1 == mag2);
```

8.9. Cache Management

These methods allow you to interact indirectly with the `PersistenceManager`'s cache.

```
public void evict (Object pc);
public void evictAll (Collection pcs);
public void evictAll (Object[] pcs);
public void evictAll ();
```

Evicting an object tells the `PersistenceManager` that your application no longer needs that object. Unless it is dirty, the object transitions to hollow and the `PersistenceManager` releases all strong references to it, allowing it to be garbage collected.

Calling the `evictAll` method with no parameters acts on all persistent-clean objects in the `PersistenceManager`'s cache.

```
public void flush ();
```

The `flush` method writes any changes that have been made in the current transaction to the datastore. If the `PersistenceManager` does not already have a connection to the datastore, it obtains one for the flush and retains it for the duration of the

transaction. Any exceptions during flush cause the transaction to be marked for rollback, as if `Transaction.setRollbackOnly` were called. See [Chapter 9, Transaction \[226\]](#)

Flushing has no effect outside of a transaction.

```
public void checkConsistency ();
```

This method validates the dirty objects in the `PersistenceManager` cache against the datastore. Within a datastore transaction, this method may delegate to `flush`. Within an optimistic transaction, this method obtains a datastore connection and checks that the instances in cache are consistent with their datastore records. If the implementation detects any inconsistencies, such as concurrent modification by another thread/process, it throws a `JDOOptimisticVerificationException`. Any datastore resources obtained during the course of this method are released before the method returns. This method has no effect outside of a transaction.

You can use `checkConsistency` to determine if a subsequent commit is likely to succeed. If `checkConsistency` throws a `JDOOptimisticVerificationException`, you have an opportunity to gather the failed objects in the exception and its nested exceptions and fix their inconsistencies before committing.

8.10. Extent Factory

```
public Extent getExtent (Class pcClass, boolean includeSubclasses);
```

Extents are logical representations of all persistent instances of a given persistence-capable class, possibly including subclasses.

Extents are obtained through the `PersistenceManager`'s `getExtent` method. This method takes two parameters: the class of objects the `Extent` contains, and a `boolean` indicating whether or not the `Extent` also contains subclass instances.

You cannot retrieve `Extents` for persistence-capable classes whose metadata specifies a value of `false` for the `requires-extent` attribute.

8.11. Query Factory

```
public Query newQuery ();
public Query newQuery (String query);
public Query newQuery (Class candidateClass);
public Query newQuery (Class candidateClass, String query);
public Query newQuery (Extent candidates);
public Query newQuery (Extent candidateClass, String query);
public Query newQuery (Class candidateClass, Collection candidates);
public Query newQuery (Class candidateClass, Collection candidates, String query);
public Query newQuery (Object query);
public Query newQuery (String language, Object query);
```

Query objects are used to find persistent objects matching certain criteria. You can obtain `Query` instances through one of the `PersistenceManager`'s several `newQuery` methods. Methods that take a query string parameter accept either a JDOQL filter or a JDOQL single-string query. See [Chapter 11, Query \[232\]](#) and the `PersistenceManager` [Javadoc](#) for details.

```
public Query newNamedQuery (String name);
```

This method retrieves a query defined in metadata by name. The returned `Query` instance is initialized with the information declared in metadata, but is fully mutable. For more information on named queries, read [Section 11.10, “Named Queries” \[250\]](#)

8.12. Sequence Factory

```
public Sequence getSequence (String name);
```

Sequences are value generators defined in metadata. This method allows you to access a sequence by name. [Chapter 16, Sequence \[318\]](#) covers sequences in detail.

8.13. Connection Access

```
public JDOConnection getConnection ();
```

The `getConnection` method returns the datastore connection in use by the `PersistenceManager`. If the `PersistenceManager` does not have a connection already, it obtains one. The returned object implements `JDOConnection`, which provides a single method:

```
Object getNativeConnection ();
```

This method returns the underlying datastore connection, free from all decorators. Using the raw connection is dangerous, and should be avoided when possible. Instead, cast the `JDOConnection` to the proper connection type. For example, in a relational JDO implementation, the `JDOConnection` will also implement `java.sql.Connection`.

If a JDO datastore transaction is in progress, then the returned connection will be transactionally consistent. If an optimistic transaction is in progress, the connection returned is only guaranteed to be transactionally consistent if the `PersistenceManager` has already flushed during the transaction. See [Section 9.1, “Transaction Types” \[226\]](#) for descriptions of optimistic and pessimistic transaction types.

Note

Kodo's `PersistenceManagers` expose a `beginStore` method you can use to ensure that a datastore transaction is in progress without flushing. See the method [Javadoc](#) for details.

Always close the returned connection (but not the underlying native connection!) before attempting any other `PersistenceManager` operations. The JDO implementation will ensure that the native connection is not released if a transaction is in progress.

Example 8.5. Using the PersistenceManager's Connection

```
import java.sql.*;
...
PersistenceManager pm = ...;
Connection conn = (Connection) pm.getConnection ();
```

```
// do JDBC stuff
conn.close ();
```

8.14. Closing

```
public boolean isClosed ();
public void close ();
```

When a `PersistenceManager` is no longer needed, call its `close` method. In an unmanaged environment, closing a `PersistenceManager` releases any resources it is using. The persistent objects managed by the `PersistenceManager` become invalid, as do any `Query`, `Extent`, and `Sequence` instances it created. Calling any method other than `isClosed` on a closed `PersistenceManager` results in a `JDOUserException`. You cannot close a `PersistenceManager` that is in the middle of a transaction in an unmanaged environment.

In a managed environment where transactions are controlled by an external `TransactionManager`, closing a `PersistenceManager` returns it to a pool, where it remains associated with the current global transaction. The `PersistenceManager` does not free its resources until the global transaction ends. In this environment, it is good practice to obtain the `PersistenceManager` from the `PersistenceManagerFactory` at the beginning of each business method, and to close it at the end of each method. The JDO implementation will ensure that you always receive the same `PersistenceManager` within the same transactional context.

Chapter 9. Transaction

Transactions are critical to maintaining data integrity. They are used to group operations into units of work that act in an all-or-nothing fashion. Transactions have the following qualities:

- *Atomicity*. Atomicity refers to the all-or-nothing property of transactions. Either every data update in the transaction completes successfully, or they all fail, leaving the datastore in its original state. A transaction cannot be only partially successful.
- *Consistency*. Each transaction takes the datastore from one consistent state to another consistent state.
- *Isolation*. Transactions are isolated from each other. When you are reading persistent data in one transaction, you cannot "see" the changes that are being made to that data in other uncompleted transactions. Similarly, the updates you make in one transaction cannot conflict with updates made in other concurrent transactions. The form of conflict resolution employed depends on whether you are using pessimistic or optimistic transactions. Both types are described later in this chapter.
- *Durability*. The effects of successful transactions are durable; the updates made to persistent data last for the lifetime of the datastore.

Together, these qualities are called the ACID properties of transactions. To understand why these properties are so important to maintaining data integrity, consider the following example:

Suppose you create an application to manage bank accounts. The application includes a method to transfer funds from one user to another, and it looks something like this:

```
public void transferFunds (User from, User to, double amnt)
{
    from.decrementAccount (amnt);
    to.incrementAccount (amnt);
}
```

Now suppose that user Alice wants to transfer 100 dollars to user Bob. No problem: you invoke your `transferFunds` method, supplying Alice in the `from` parameter, Bob in the `to` parameter, and `100.00` as the `amnt`. The first line of the method is executed, and 100 dollars is subtracted from Alice's account. But then, something goes wrong. An unexpected exception occurs, or the network fails, and your method never completes.

You are left with a situation in which the 100 dollars has simply disappeared. Thanks to the first line of your method, it is no longer in Alice's account, and yet it was never transferred to Bob's account either. The datastore is in an inconsistent state.

The importance of transactions should now be clear. If the two lines of the `transferFunds` method had been placed together in a transaction, it would be impossible for only the first line to succeed - either the funds would be transferred properly or they would not be transferred at all and an exception would be thrown. Money could never vanish into thin air, and the data store could never get into an inconsistent state.

9.1. Transaction Types

There are two major types of transactions: pessimistic transactions and optimistic transactions. Each type has both advantages and disadvantages.

Pessimistic transactions generally lock the datastore records they act on, preventing other concurrent transactions from using the same data. This avoids conflicts between transactions, but consumes a lot of database resources. Additionally, locking records can result in *deadlock*, a situation in which two transactions are both waiting for the other to release its locks before completing. The results of a deadlock are datastore-dependent; usually one transaction is forcefully rolled back after some specified timeout interval, and an exception is thrown.

JDO does not have pessimistic transactions per se. Instead, JDO has *datastore transactions*. This is JDO's way of admitting that some datastores do not support pessimistic semantics, and that the exact meaning of a non-optimistic JDO transaction is dependent on the datastore. In most JDO implementations, a datastore transaction is equivalent to a pessimistic transaction.

Optimistic transactions consume less resources than pessimistic/datastore transactions, but only at the expense of reliability. Because optimistic transactions do not lock datastore records, two transactions might change the same persistent information at the same time, and the conflict will not be detected until the second transaction attempts to flush or commit. At this time, the second transaction will realize that another transaction has concurrently modified the same records (usually through a timestamp or versioning system), and will throw an appropriate exception. Note that optimistic transactions still maintain data integrity; they are simply more likely to fail in heavily concurrent situations.

Despite their drawbacks, optimistic transactions are the best choice for most applications. They offer better performance, better scalability, and lower risk of hanging due to deadlock.

Note

Kodo supports both optimistic and datastore transactions. Kodo defaults JDO datastore transactions to pessimistic semantics; however, Kodo also offers advanced locking and versioning APIs for fine-grained control over database resource allocation and object versioning. See [Section 9.4, “Object Locking” \[542\]](#) and [Section 5.8, “Lock Groups” \[476\]](#) of the Reference Guide for details on locking. [Section 15.10, “Version” \[284\]](#) of this document covers object versioning strategies.

9.2. The JDO Transaction Interface

javax.jdo
Transaction
- Synchronization: Synchronization
- NonTransactionalRead: boolean
- NonTransactionalWrite: boolean
- RetainValues: boolean
- RestoreValues: boolean
- Optimistic: boolean
- begin()
- commit()
- rollback()
- setRollbackOnly()
- getRollbackOnly(): boolean
- isActive(): boolean

The `Transaction` interface controls transactions in JDO. This interface consists of "getter" and "setter" methods for several Java bean-style properties and standard transaction demarcation methods.

Note

Kodo offers additional transaction-related functionality through the `KodoPersistenceManager`.

Kodo can also integrate with your application server's JTA service. See [Section 8.2, “Integrating with the Transaction Manager” \[535\]](#) in the Reference Guide for details.

9.2.1. Transaction Properties

```
public boolean getNontransactionalRead ();
public void setNontransactionalRead (boolean read);
public boolean getNontransactionalWrite ();
```

```

public void setNontransactionalWrite (boolean write);
public boolean getRetainValues ();
public void setRetainValues (boolean retain);
public boolean getRestoreValues ();
public void setRestoreValues (boolean restore);
public boolean getOptimistic ();
public void setOptimistic (boolean optimistic);
public Synchronization getSynchronization ();
public void setSynchronization (Synchronization synch);

```

The Transaction's `NontransactionalRead`, `NontransactionalWrite`, `RetainValues`, `RestoreValues`, and `Optimistic` properties mirror those presented in [Section 7.2.2, “PersistenceManager and Transaction Defaults” \[209\]](#). We need not discuss them again.

The final Transaction property, `Synchronization`, has not been covered yet. This property enables you to associate a `javax.transaction.Synchronization` instance with the Transaction. The Transaction will notify your `Synchronization` instance on transaction completion events, so that you can implement custom behavior on commit or rollback. See the `javax.transaction.Synchronization` Javadoc for details.

Note

Kodo implements a complete transaction event listener framework in addition to the standard `Synchronization` callbacks mandated by the JDO specification. See the `kodo.event` package [Javadoc](#) for details.

9.2.2. Transaction Demarcation

```

public void begin ();
public void commit ();
public void rollback ();

```

The `begin`, `commit`, and `rollback` methods demarcate transaction boundaries. The methods should be self-explanatory: `begin` starts a transaction, `commit` attempts to commit the transaction's changes to the datastore, and `rollback` aborts the transaction, in which case the datastore is "rolled back" to its previous state. JDO implementations will automatically roll back transactions if any exception is thrown during the commit process.

```

public void setRollbackOnly ();
public boolean getRollbackOnly ();

```

In a large system, it is generally good practice to isolate transaction demarcation code from business logic. Your business methods should not begin and end transactions; that should be left to the calling code. In fact, declarative transaction frameworks like EJB's Container Managed Transactions (CMT) get rid of explicit transaction demarcation altogether.

Sometimes, though, your business methods may encounter an error that invalidates the current transaction. The `setRollbackOnly` method allows your business logic to mark the transaction for rollback. The method takes no parameters: once a transaction has been marked for rollback it cannot be unmarked. A transaction that has been marked for rollback cannot commit successfully. A call to `commit` will result in an exception and an automatic rollback. The `getRollbackOnly` method allows you to test whether the current transaction has been marked for rollback.

```

public boolean isActive ();

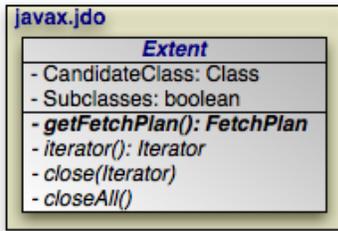
```

Finally, the `isActive` method returns `true` if the transaction is in progress (`begin` has been called more recently than `commit` or `rollback`), and `false` otherwise.

Example 9.1. Grouping Operations with Transactions

```
public void transferFunds (User from, User to, double amnt)
{
    // note: it would be better practice to move the transaction demarcation
    // code out of this method, but for the purposes of example...
    PersistenceManager pm = JDOHelper.getPersistenceManager (from);
    Transaction trans = pm.currentTransaction ();
    trans.begin ();
    try
    {
        from.decrementAccount (amnt);
        to.incrementAccount (amnt);
        trans.commit ();
    }
    catch (JDOFatalException jfe) // trans is already rolled back
    {
        throw jfe;
    }
    catch (RuntimeException re) // includes non-fatal JDO exceptions
    {
        trans.rollback (); // or could attempt to fix error and retry
        throw re;
    }
}
```

Chapter 10. Extent



An Extent is a logical view of all persistent instances of a given persistence-capable class, possibly including subclasses. Extents are obtained from PersistenceManagers, and are usually used to specify the candidate objects to a Query.

Note

Kodo extends the standard Extent with the **KodoExtent** interface to provide additional functionality. You can cast any Extent in Kodo to a KodoExtent.

```
public Class getCandidateClass ();
public boolean hasSubclasses ();
```

The `getCandidateClass` method returns the persistence-capable class of the Extent's instances. The `hasSubclasses` method indicates whether instances of subclasses are part of the Extent as well.

```
public FetchPlan getFetchPlan ();
```

Before iterating an Extent, you may want to use the FetchPlan to optimize which fields and relations of the returned objects will be loaded, and to configure result scrolling. See [Chapter 12, FetchPlan \[254\]](#) for details on the FetchPlan interface.

When you create an Extent with `PersistenceManager.getExtent`, the Extent's FetchPlan is initialized to the same values as the plan of its owning PersistenceManager. Subsequent changes to the Extent's FetchPlan, however, will not affect the PersistenceManager's plan.

```
public Iterator iterator ();
public void close (Iterator itr);
public void closeAll ();
```

You can obtain an iterator over every object in an Extent using the `iterator` method. The iterators used by some implementations might consume datastore resources; therefore, you should always close an Extent's iterators as soon as you are done with them. You can close an individual iterator by passing it to the `close` method, or all open iterators at once with `closeAll`.

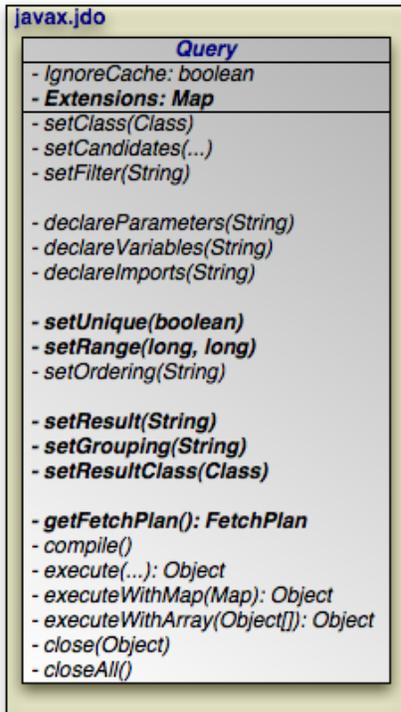
Note

Kodo may use scrollable JDBC `ResultSets` when large data sets are being iterated. Combined with Kodo's memory-sensitive data structures, this allows you to efficiently iterate over huge data sets - even when the entire data set could not possibly fit into memory at once. These scrollable results consume database resources, however, so you are strongly encouraged to close your iterators when you are through with them. If they are not closed immediately, they will be closed when they are garbage collected by the JVM.

Example 10.1. Iterating an Extent

```
PersistenceManager pm = ...;
Extent mags = pm.getExtent (Magazine.class, true);
Iterator itr = employees.iterator ();
while (itr.hasNext ())
    processMagazine ((Magazine) itr.next ());
mags.close (itr);
```

Chapter 11. Query



The `javax.jdo.Query` interface serves two functions: object filtering and data retrieval. This chapter examines each in turn.

Note

Kodo extends the standard `Query` interface with the `KodoQuery` interface to provide additional functionality. You can cast any `Query` in Kodo to a `KodoQuery`.

11.1. Object Filtering

JDO queries evaluate a group of candidate objects against a set of conditions, eliminating the objects that don't match. We refer to this as *object filtering*. The original group of objects might be a `Collection` of instances you've already retrieved, or an `Extent`. Recall from **Chapter 10, *Extent*** [230] that an `Extent` represents all persistent instances of a certain class, optionally including subclasses.

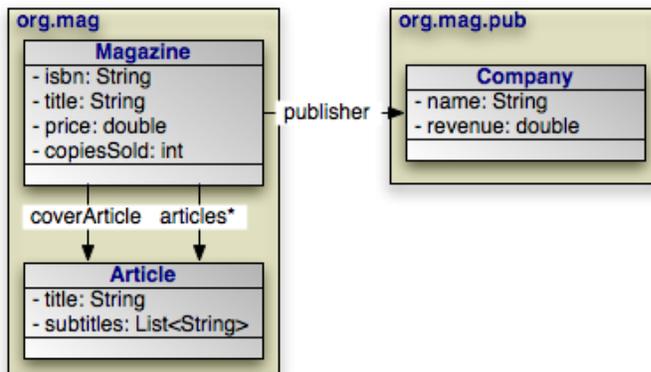
Filter conditions are specified in the JDO Query Language (JDOQL). The filtering process might take place in the datastore, or might be executed in memory. JDO does not mandate any one mechanism, and many implementations use a mixture of datastore and in-memory execution depending on the circumstances.

Basic object filtering utilizes the following `Query` methods:

```
public void setClass (Class candidateClass);
public void setCandidates (Extent candidates);
public void setCandidates (Collection candidates);
public void setFilter (String filter);
```

- `setClass` names the candidate class. Candidate objects that are not assignable to the candidate class cannot match the filter.
- The two `setCandidates` methods specify the set of objects to evaluate. If you supply an `Extent`, you don't need to also call `setClass`; the `Query` inherits the `Extent`'s candidate class. Reciprocally, if you call `setClass` but do not call either `setCandidates` method, the query candidates default to the `Extent` of the candidate class, including subclasses.
- `setFilter` accepts a JDOQL string establishing the conditions an object must meet to be included in the query results. As you'll see shortly, JDOQL looks exactly like a Java boolean expression using the candidate class' persistent fields and relations. When you don't set a filter explicitly, it defaults to the simplest possible boolean expression: `true`. In this case, all candidate objects assignable to the candidate class match the query.

Let's see an example of basic object filtering in action. Our example draws on the following subset of the object model defined in **Chapter 5, *Metadata*** [193]. We use this model subset throughout this chapter.



Example 11.1. Filtering

The following code processes all `Magazine` objects in the database whose price is greater than 10 dollars.

```

PersistenceManager pm = ...;
Query query = pm.newQuery ();
query.setClass (Magazine.class);
query.setCandidates (pm.getExtent (Magazine.class, true));
query.setFilter ("this.price > 10");
List mags = (List) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
  
```

The code above is technically correct, but could be much more concise. First, remember that setting a candidate class defaults the candidates to the `Extent` of that class, and vice versa. So we don't need both the `setClass` and `setCandidates` calls. We can also get rid of the `this` qualifier on `price` in our filter string, since unqualified field names are already assumed to belong to the current object, just as in Java code. Finally, we can take advantage of the fact that `PersistenceManager` overloads the `newQuery` method to tighten our code further. Here is the revised version:

```

PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class, "price > 10");
List mags = (List) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
  
```

The filter above is rather basic. Before moving on to more advanced queries, though, we need to dive into the details of JDOQL.

11.2. JDOQL

JDOQL is a datastore-neutral query language based on Java. It relies solely on your object model, completely abstracting the details of the underlying data model. As you have already seen, the syntax of JDOQL is virtually identical to Java syntax. In fact, a good trick when writing a JDOQL string is to pretend that you are writing a method of the query's candidate class. You have access to all of the class' persistent fields, and to the `this` keyword. There are, however, some notable ways in which JDOQL differs from Java:

- You can traverse through private persistent fields of relations.
- String literals can be either double-quoted or single-quoted.
- Equality and ordering comparisons between primitives and instances of wrapper classes (`Boolean`, `Byte`, `Integer`, etc) are valid.
- Equality and ordering between `Dates` are valid.
- Equality comparisons always use the `==` operator; the `equals` method is not supported.
- The assignment operators (`=`, `+=`, `*=`, etc) as well as the `++` and `--` operators are not supported.
- Methods are not supported, with the following exceptions:
 - `Collection.contains`
 - `Collection.isEmpty`
 - `Map.containsKey*`
 - `Map.containsValue*`
 - `Map.isEmpty*`
 - `String.toUpperCase*`
 - `String.toLowerCase*`
 - `String.startsWith`
 - `String.endsWith`
 - `String.matches*`: Regular expression support is limited to the use of `.`, `*`, and `(?i)` to represent case-insensitive matching.
 - `String.indexOf*`: Both the `String.indexOf(String str)` and the `String.indexOf(String str, int fromIndex)` forms are supported.
 - `String.substring*`: Both the `String.substring(int start)` and the `String.substring(int beginIndex, int endIndex)` forms are supported.
 - `Math.abs*`
 - `Math.sqrt*`
 - `JDOHelper.getObjectId*`

- JDOQL supports the following aggregates: `min`, `max`, `sum`, `avg`, `count*`. We discuss aggregates in detail later in the chapter.
- Traversing a null-valued field causes the subexpression to evaluate to `false` rather than throwing a `NullPointerException`.

Note

Kodo offers several extensions to JDOQL, and allows you to define your own extensions. See the Reference Guide's [Section 9.6, “Query Language Extensions” \[550\]](#) and [Section 9.6.4, “JDOQL Subqueries” \[553\]](#) for details.

While it is important to note the differences between JDOQL and Java enumerated in the list above, it is just as important to note what is *not* in the list. Mathematical operators, logical operators, `instanceof`, casting, field traversal, and static field access are omitted, meaning they are all fully supported by JDOQL.

We demonstrate some of the interesting aspects of JDOQL below.

Example 11.2. Relation Traversal and Mathematical Operations

Find all magazines whose sales account for over 1% of the total revenue for the publisher. Notice the use of standard Java "dot" notation to traverse into the persistent fields of `Magazine`'s `publisher` relation.

```
Query query = pm.newQuery (Magazine.class,
    "price * copiesSold > publisher.revenue * .01");
List mags = (List) query.execute ();
```

Example 11.3. Precedence, Logical Operators, and String Functions

Find all magazines whose publisher's name is "Random House" or matches a regular expression, and whose price is less than or equal to 10 dollars. Here, we use single-quoted string literals to avoid having to escape double quotes within the filter string. Notice also that we compare strings with `==` rather than the `equals` method.

```
Query query = pm.newQuery (Magazine.class, "price <= 10.0 "
    + "&& (publisher.name == 'Random House' "
    + "|| publisher.name.toLowerCase ().matches ('add.*ley'))");
List mags = (List) query.execute ();
```

Example 11.4. Collections

Find all magazines whose cover article has a subtitle of "The Real Story" or whose cover article has no subtitles.

```
Query query = pm.newQuery (Magazine.class,
    "coverArticle.subtitles.contains ('The Real Story') "
    + "|| coverArticle.subtitles.isEmpty ()");
List mags = (List) query.execute ();
```

Example 11.5. Static Methods

Retrieve all magazines whose publisher's revenue's square root is greater than 100 dollars.

```
Query query = pm.newQuery (Magazine.class,
    "Math.sqrt (publisher.revenue) > 100");
List mags = (List) query.execute ();
```

11.3. Advanced Object Filtering

In this section, we explore advanced topics in object filtering, supported by the following `Query` methods:

```
public void declareParameters (String parameters);
public void declareVariables (String variables);
public void declareImports (String imports);
```

- The `declareParameters` method defines the names and types of the query's parameters. Declaring query parameters is analogous to declaring the parameter signature of a Java method, and uses the same syntax. Parameters acts as placeholders in the filter string, allowing you to supply new values on each execution. Parameters also allow for *query by example*, as demonstrated in **Example 11.8, “Query By Example” [238]**

Parameters do not have to be declared. As you will see in the examples below, you can introduce *implicit* parameters into your queries simply by prefixing the parameter name with a colon in your JDOQL string. Wherever the type of an implicit parameter can't be inferred from the context, you can set the type by casting the parameter in the JDOQL.

You cannot mix implicit and declared parameters in the same query. Each query must declare all of its parameters, or none of them.

- `declareVariables` names the local variables used in the query filter. It uses standard Java variable declaration syntax. Query variables are typically used to place conditions on elements of collections or maps.

Like parameters, variables do not have to be declared. Whenever you use an unrecognized identifier where a variable would be appropriate, the JDO implementation will dynamically define an *implicit* variable with that name. In the case of an unconstrained variable, you must cast the variable in your JDOQL to supply its type. Unlike parameters, you can mix both declared and implicit variables in the same query.

- You can import classes into the query's namespace with the `declareImports` method. The method argument uses standard Java `import` syntax. By default, queries only recognize unqualified class names when the class is in the package of the candidate class, or when it is in `java.lang`. Using imports, you can save yourself the trouble of typing out full class names in your parameter and variable declarations, and in your your filter string (class names can appear in filter strings when you use the `instanceof` operator, access a static field, or perform a cast).

The following examples will give you a feel for the query elements described above. They use the object model we defined in the **previous section**.

Example 11.6. Imports and Declared Parameters

Find the magazines with a certain publisher and title, where the publisher and title are supplied as parameters on each execution.

```
PersistenceManager pm = ...;
Company comp = ...;
String str = ...;

Query query = pm.newQuery (Magazine.class,
    "publisher == pub && title == ttl");
query.declareImports ("import org.mag.pub.*");
query.declareParameters ("String ttl, Company pub");
List mags = (List) query.execute (str, comp);
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

There are two things to take away from this example. First, the import prevents us from having to qualify the `Company` class name when declaring our `pub` parameter (although in this case, it would have been easier to just write out the full class name in the parameter declaration). The unqualified `Company` name is not automatically recognized by the query because `Company` isn't in the same package as `Magazine`, the candidate class.

Second, notice that we supply values for the parameter placeholders when we execute the query. Parameters can be of any type recognized by JDO, though primitive parameters have to be supplied as instances of the appropriate wrapper type on execution. The `Query` interface includes several methods for executing queries with various numbers of parameters. When you use declared parameters, you should supply the parameter values in the same order that you declare the parameters.

Example 11.7. Implicit Parameters

The query below is exactly the same as our previous example, except this time we use implicit parameters.

```
PersistenceManager pm = ...;
Company comp = ...;
String str = ...;

Query query = pm.newQuery (Magazine.class,
    "publisher == :pub && title == :ttl");
List mags = (List) query.execute (comp, str);
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

Here, we use colon-prefixed names to introduce new parameters without declarations. When we execute the query, we supply the parameter values in the order the implicit parameters first appear in the JDOQL. Later in the chapter, we'll see queries that consist of multiple JDOQL strings. Each string might introduce new implicit parameters. When deciding the proper order to supply the parameter values, start with the parameters in the result string, then those in the filter, then the grouping string, and finally the ordering string.

In the query above, we can infer the type of each parameter based on its context. This is almost always the case. There are times, however, when the context alone is not enough to determine the type of an implicit parameter. In these cases, use a cast to supply the parameter type:

```
PersistenceManager pm = ...;
Company comp = ...;

Query query = pm.newQuery (Magazine.class,
    "publisher.revenue == ((org.mag.pub.Company) :pub).revenue");
List mags = (List) query.execute (comp);
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

Example 11.8. Query By Example

Parameters do not need to be in the datastore to be useful. You can implement *query by example* in JDO by using an existing "example" object as a query parameter:

```
PersistenceManager pm = ...;
Magazine example = new Magazine ();
example.setPrice (100);
example.setTitle ("Fourier Transforms");
Query query = pm.newQuery (Magazine.class,
    "price == ex.price && title == ex.title");
query.declareParameters ("Magazine ex");
List mags = (List) query.execute (example);
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
```

Example 11.9. Variables

Find all magazines that have an article titled "Fourier Transforms".

```
PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class, "articles.contains (art) "
    + "&& art.title == 'Fourier Transforms'");
query.declareVariables ("Article art");
List mags = (List) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

A variable represents any persistent instance of its declared type. So you can read the filter string above as: "The magazine's `articles` collection contains some article `art`, where `art`'s title is 'Fourier Transforms'". Notice how we bind `art` to a particular collection with the `contains` method, then test its properties in an `&&`'d expression. This is a common pattern in JDOQL filters, and applies equally well to placing conditions on the keys and values of maps.

Of course, we don't have to declare `art` explicitly. The same query without `declareVariables` would work just as well:

```
PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class, "articles.contains (art) "
    + "&& art.title == 'Fourier Transforms'");
List mags = (List) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

In this case the JDO implementation assumes `art` is an implicit variable because it does not match the name of any field in the candidate class. The new variable's type is set to the known element type of the collection that contains it.

Example 11.10. Unconstrained Variables

The example above uses a variable to represent any element in a collection. We refer to variables used to test collection or map

elements as *constrained* or *bound* variables, because the values of the variable are limited by the collection or map involved. Many JDO implementations also support *unconstrained* variables. Rather than representing a collection or map element, an unconstrained variable represents any persistent instance of its class. Consider the following example:

```
Query query = pm.newQuery (Article.class, "mag.copiesSold > 10000 "
    + "&& mag.coverArticle == this");
query.declareVariables ("Magazine mag");
List arts = (List) query.execute ();
for (Iterator itr = arts.iterator (); itr.hasNext ();)
    processArticle ((Article) itr.next ());
query.close (arts);
```

What does this query do? Let's break it down. The first clause matches any magazine that has sold more than 10,000 copies. The second clause requires that the cover article of the magazine is the candidate instance being evaluated (notice the query's candidate class is `Article` in this example). So the query returns all articles that are the cover article for a magazine that has sold more than 10,000 copies. The unconstrained variable `mag` allowed us to overcome the fact that there was no direct relation from `Article` to `Magazine` (only the reverse).

Later in this chapter, we'll see how to use a *projection* to drastically simplify this query.

We can also execute this query without declaring the `mag` variable explicitly. Without a constraining `contains` clause, however, the type of an unconstrained, implicit variable is impossible to infer. Use a cast to supply the type:

```
Query query = pm.newQuery (Article.class, "((Magazine) mag).copiesSold > 10000 "
    + "&& mag.coverArticle == this");
List arts = (List) query.execute ();
for (Iterator itr = arts.iterator (); itr.hasNext ();)
    processArticle ((Article) itr.next ());
query.close (arts);
```

Up until now, we have focused on how to configure our queries with the right filter, but we have ignored how to actually execute the query. The next section corrects this oversight.

11.4. Compiling and Executing Queries

```
public FetchPlan getFetchPlan ();
```

Before compiling and/or executing your query, you may want to use the query's `FetchPlan` to optimize which fields and relations of your result objects will be loaded, and to configure result scrolling. See [Chapter 12, *FetchPlan* \[254\]](#) for details on the `FetchPlan` interface.

When you create a query with `PersistenceManager.newQuery` or `PersistenceManager.newNamedQuery`, the query's `FetchPlan` is initialized to the same values as the plan of its owning `PersistenceManager`. Subsequent changes to the query's `FetchPlan`, however, will not affect the `PersistenceManager`'s plan.

```
public void compile ();
```

Compiling is a hint to the implementation to optimize an execution plan for the query. During compilation, the query validates all of its elements and reports any inconsistencies by throwing a `JDOUserException`. Queries automatically compile themselves

before they execute, so you probably won't use this method often.

```
public Object execute ();
public Object execute (Object param1);
public Object execute (Object param1, Object param2);
public Object execute (Object param1, Object param2, Object param3);
public Object executeWithArray (Object[] params);
public Object executeWithMap (Map params);
```

As evident from the examples throughout this chapter, queries are executed with one of the many `execute` methods defined in the `Query` interface. If your query uses between 0 and 3 parameters, you can use the `execute` version that takes the proper number of `Object` arguments. For queries with more than 3 parameters, you can use the generic `executeWithArray` and `executeWithMap` methods. In the latter method, the map is keyed on the parameter name.

All `execute` methods declare a return type of `Object` because the return type depends on how you've configured the query. So far we've only seen queries that return `Lists`, but this will change in the next section.

```
public void close (Object result);
public void closeAll ();
```

Some query results may hold on to datastore resources. Therefore, you should close query results when you no longer need them. You can close an individual query result with the `close` method, or all open results at once with `closeAll`.

Note

By default, Kodo greedily fills each result list and immediately releases all database resources, making `close` unnecessary. You can, however, configure Kodo to use lazy element instantiation where appropriate. Under lazy instantiation, list elements that are never accessed are never created, and elements may be released after being traversed. Combined with Kodo's memory-sensitive data structures and smart eager fetching policies, lazy element instantiation also allows you to efficiently iterate over huge data sets - even when the entire data set could not possibly fit into memory at once. These lazy results consume database resources, so if you enable them you are strongly encouraged to close all query results after processing. If they are not closed explicitly, they are closed automatically when the JVM garbage collects them. See [Section 4.10, "Large Result Sets" \[440\]](#) in the Reference Guide for details on configuring Kodo for lazy result instantiation.

11.5. Limits and Ordering

We have seen how you tell the query *which* objects you want. This section shows you how to also tell it *how many* objects you want, and *what order* you want them in.

```
public void setUnique (boolean unique);
public void setRange (long start, long end);
```

Use the `setUnique` method when you know your query matches at most a single object. A unique query always returns either the one candidate instance that matched the filter, or null; it never returns a `List`. In fact, unique queries enforce the one-result rule by throwing a `JDOUserException` if more than one candidate survives the filtering process.

While `setUnique` is designed for queries with only one result, `setRange` is designed for those with many. Most applications that deal with large amounts of data only process or display a fixed number of records at a time. For example, a web app might show the user 20 results per page, though many thousands of records exist in the database. Using `setRange`, you can retrieve

the exact range of objects you're interested in. The method behaves much like `List.subList` or `String.substring`: it uses 0-based indexes, and the end index is exclusive. Unlike these methods, however, `setRange` won't throw an exception when your range extends beyond the available elements. If 50 records exist and you ask for the range 40, 60, you'll receive a list containing the 41st (remember indexes are 0-based) through 50th result objects. If you ask for the range 60, 80, you'll get an empty list.

Whenever you don't specify a range on a query, it defaults to a start index of 0 and an end index of `Long.MAX_VALUE`, which represents no limit.

```
public void setOrdering (String ordering);
```

Ranges are meaningless if the result order isn't constant between query executions, and many datastores don't guarantee consistent natural ordering. `setOrdering` gives you control over the order of results, so you don't have to rely on the vagaries of the datastore. The method's argument is a comma-separated list of field names or expressions, each followed by the keyword `ascending` or `descending`. Those keywords can be abbreviated as `asc` and `desc` respectively. Results are ordered primarily by the first (left-most) expression. Wherever two results compare equal with that expression, the next ordering expression is used to order them, and so on.

Example 11.11. Unique

Find the magazine "Spaces" published by Manning. We know that this query can only match 0 or 1 magazines, so we set the `unique` flag to avoid having to extract the result object from a `List`.

```
PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class,
    "publisher.name == 'Manning' && title == 'Spaces'");
query.setUnique (true);
processMagazine ((Magazine) query.execute ());
```

Example 11.12. Result Range and Ordering

Order all magazines by price and return the 21st through 40th results.

```
PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class);
query.setOrdering ("price ascending");
query.setRange (20, 40);
List mags = (List) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
query.close (mags);
```

11.6. Projections

At the beginning of the chapter, we said that JDO queries serve two purposes: object filtering and data retrieval. So far, we have concentrated primarily on object filtering. We now shift our focus to data retrieval, starting with projections.

A *projection* consists of one or more values formed by traversing the fields of the candidates. You can use projections to get back only the fields or relations that you are interested in, rather than manually pulling the desired data from each result object. As you will see later, you can even instruct JDO to pack the projected field values into a class of your choosing.

Projections are established with the `setResult` method:

```
public void setResult (String result);
```

`result` is a comma-separated string of result expressions. Each result expression might be a field name, a relation traversal, or any other valid JDOQL clause. In all our queries to this point, we've taken advantage of the fact that when you don't specify a result string explicitly, it defaults to `this`, meaning the query returns each matching object itself (actually, `result` defaults to `distinct this as C`, where `C` is the unqualified candidate class name, but for now just think of it as `this`).

We present a simple projection below. We're still using the object model defined in [Section 11.1, "Object Filtering"](#) [232].

Example 11.13. Projection

```
Query query = pm.newQuery (Magazine.class);
query.setResult ("title, price");
List results = (List) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    Object[] data = (Object[]) itr.next ();
    processData ((String) data[0], (Double) data[1]);
}
query.close (results);
```

Each query result in this case is not a `Magazine` instance, but an array consisting of each magazine's title and price. Projections allow you to acquire only the data you're interested in, and ignore the rest.

`JDOHelper.getObjectId` becomes a particularly useful JDOQL method when applied to projections. Notice that because we only select a single result expression this time, we get back the raw result values (in this case, identity objects) instead of object arrays.

```
Query query = pm.newQuery (Magazine.class, "publisher.revenue > 1000000");
query.setResult ("JDOHelper.getObjectId (this)");
List oids = (List) query.execute ();
for (Iterator itr = oids.iterator (); itr.hasNext ();)
    processObjectId (itr.next ());
query.close (oids);
```

Projections aren't limited to the candidate class and its fields. Here is a query selecting the first 3 characters of the cover article's title for all magazines under 5 dollars:

Example 11.14. Projection Field Traversal

```
Query query = pm.newQuery (Magazine.class, "price < 5");
query.setResult ("coverArticle.title.substring (0, 3)");
List ttls = (List) query.execute ();
for (Iterator itr = ttls.iterator (); itr.hasNext ();)
    processTitle ((String) itr.next ());
```

```
query.close (ttls);
```

But what if we change our minds, and decide we're really interested in the *subtitles* of all cover articles, rather than their titles? Setting the result query to `coverArticle.subtitles.substring (0, 3)` won't do the trick; `subtitles` is a `Collection` field, and `Collection` doesn't have a `substring` method. What we're after isn't the `subtitles` field itself, but its elements.

You may recall from previous examples that we used query variables to represent the elements of collections. We can apply the same solution here. By binding a variable to the elements of the `subtitles` collection in our filter and selecting the variable in our result string, we achieve our goal.

Example 11.15. Projection Variables

```
Query query = pm.newQuery (Magazine.class, "price < 5 "
    + "&& coverArticle.subtitles.contains (ttl)");
query.setResult ("ttl.substring (0, 3)");
List ttls = (List) query.execute ();
for (Iterator itr = ttls.iterator (); itr.hasNext ();)
    processSubtitle ((String) itr.next ());
query.close (ttls);
```

Now we have our subtitle data, but we're faced with a new problem: a lot of subtitles start with the same 3 characters, and we really only want to process each string once. Luckily, JDO has a built-in solution. If you begin a result string with the `distinct` keyword, JDO filters out duplicates. This applies not only to simple projections as in this example, but to complex projections consisting of multiple return values, including persistent object values.

Example 11.16. Distinct Projection

```
Query query = pm.newQuery (Magazine.class, "price < 5 "
    + "&& coverArticle.subtitles.contains (ttl)");
query.setResult ("distinct ttl.substring (0, 3)");
List ttls = (List) query.execute ();
for (Iterator itr = ttls.iterator (); itr.hasNext ();)
    processSubtitle ((String) itr.next ());
query.close (ttls);
```

Note

Some object-relational mapping products require you to alter your query to circumvent duplicate results caused by relational joins. In the simple case, this may only involve using a `SELECT DISTINCT` equivalent. But in the complex case, such as with aggregate data, you might need to tell the implementation to use subselects and other complicated workarounds to avoid the relational joins problem.

This is not the case in JDO. JDO implementations always automatically eliminate duplicates caused by relational joins. You only need to use the `distinct` keyword in your JDO result string when there are repeated values in the database that you'd like to filter out.

Remember the unconstrained variable example earlier in this chapter? We wanted to find all the cover articles of magazines that sold over 10,000 copies, but we had to work around the fact that there is no relation from `Article` to `Magazine`:

```
Query query = pm.newQuery (Article.class, "mag.copiesSold > 10000 "
    + "&& mag.coverArticle == this");
query.declareVariables ("Magazine mag");
List arts = (List) query.execute ();
for (Iterator itr = arts.iterator (); itr.hasNext ());
    processArticle ((Article) itr.next ());
query.close (arts);
```

To conclude the section, let's simplify this query using our newfound knowledge of projections:

```
Query query = pm.newQuery (Magazine.class, "copiesSold > 10000");
query.setResult ("coverArticle");
List arts = (List) query.execute ();
for (Iterator itr = arts.iterator (); itr.hasNext ());
    processArticle ((Article) itr.next ());
query.close (arts);
```

11.7. Aggregates

Aggregates are just what they sound like: aggregations of data from multiple instances. Combined with object filtering and grouping, aggregates are powerful tools for summarizing your persistent data.

JDOQL includes the following aggregate functions:

- `min(expression)`: Returns the minimum value of the given expression among matching instances.
- `max(expression)`: Returns the maximum value of the given expression among matching instances.
- `sum(expression)`: Returns the sum of the given expression over all matching instances.
- `avg(expression)`: Returns the average of the given expression over all matching instances.
- `count(expression)`: Returns the number of matching instances for which the given expression is not null.

Note

Kodo allows you to define your own aggregate functions. See [Section 9.6, “Query Language Extensions” \[550\]](#) in the Reference Guide for details.

The following example counts the number of magazines that cost under 5 dollars:

Example 11.17. Count

```
Query query = pm.newQuery (Magazine.class, "price < 5");
query.setResult ("count(this)");
Long count = (Long) query.execute ();
```

You may be thinking that we could have gotten the count just as easily by executing a standard query and calling `Collection.size()` on the result, and you'd be right. But not all aggregates are so easy to replace. Our next example retrieves the minimum, maximum, and average magazine prices in the database. These values would be a little more difficult to calculate manually. More importantly, iterating over every single persistent magazine in order to factor its price into our calculations would be woefully inefficient.

Example 11.18. Min, Max, Avg

```
Query query = pm.newQuery (Magazine.class);
query.setResult ("min(price), max(price), avg(price)");
Object[] prices = (Object[]) query.execute ();
Double min = (Double) prices[0];
Double max = (Double) prices[1];
Double avg = (Double) prices[2];
```

The functionality described above is useful, but aggregates only really shine when you combine them with object grouping.

```
public void setGrouping (String grouping);
```

The `Query` interface's `setGrouping` method allows you to group query results on field values. The grouping string consists of one or more comma-separated clauses to group on, optionally followed by the `having` keyword and a boolean expression. The `having` expression pares down the candidate groups just as the query's filter pares down the candidate objects.

Now your aggregates apply to *each group*, rather than to all matching objects. Let's see this in action:

Example 11.19. Grouping

The following query returns each publisher and the average price of its magazines, for all publishers that publish no more than 10 magazines.

```
Query query = pm.newQuery (Magazine.class);
query.setResult ("publisher, avg(price)");
query.setGrouping ("publisher having count(this) <= 10");
List results = (List) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    Object[] data = (Object[]) itr.next ();
    processData ((Company) data[0], (Double) data[1]);
}
query.close (results);
```

You probably noticed that in our initial aggregate examples, the queries all returned a single result object, while the query above returned a `List`. Before you get too confused, let's take a brief detour to examine query return types.

- If you have called `setUnique(true)`, the query returns a single result object (or null).
- Else if you have called `setUnique(false)`, the query returns a `List`.

- Else if the query result is an aggregate and you have not specified any grouping, the query returns a single result object.
- Else the query returns a `List`.

In addition to how many results are returned, query configuration can affect the type of each result:

- If you have specified a result class, the query returns instances of that class. We cover result classes in the next section.
- Else if you have not set a result string, the query returns instances of the candidate class.
- Else if you have specified a single projection or aggregate result:
 - A projection returns instances of the projected field type. When mathematical expressions are involved, the less precise operand is always promoted to the type of the more precise operand.
 - `min(expression)` returns an instance of the expression type.
 - `max(expression)` returns an instance of the expression type.
 - `sum(expression)` returns a `long` for integral types other than `BigInteger`, and the expression type for all other types.
 - `avg(expression)` returns an instance of the expression type.
 - `count(expression)` returns a `long`.
- Else if you have specified multiple projections or aggregates in your result string, the query returns instances of `Object[]`, where the class of each array index value follows the typing rules above.
- Casting a projection or aggregate expression in the result string converts that result element to the type specified in the cast.

Don't worry about trying to memorize all of these rules. In practice, they amount to a much simpler rule: queries return what you expect them to. A query for all the magazines that match a filter returns a list of `Magazines`. But an aggregate query for the count of all magazines that match a filter just returns a `Long`. A projection query for the title of all magazines returns a list of `Strings`. But a projection for both the title and price of each magazine returns a list of `Object[]`s, each consisting of a `String` and a `Double`. So although you can always explicitly set the `unique` flag and result class to obtain a specific result shape, the defaults are usually exactly what you want.

11.8. Result Class

Queries have the ability to pack result data into a custom result class. You might use this feature for anything from populating data transfer objects automatically, to avoiding the casting and other inconveniences involved in dealing with the `Object[]`s normally generated by multi-valued projections. You specify a custom result class with the `setResultClass` method.

```
public void setResultClass (Class resultClass);
```

11.8.1. JavaBean Result Class

JDO populates result objects by matching the result class' public fields and JavaBean setter methods to the expressions defined in the query result string. The result class must be public (or otherwise instantiable via reflection), and must have a no-args constructor.

Example 11.20. Populating a JavaBean

```

public class SalesData
{
    private double price;
    private int copiesSold;

    public void setPrice (double price)
    {
        this.price = price;
    }

    public double getPrice ()
    {
        return price;
    }

    public void setCopiesSold (int copiesSold)
    {
        this.copiesSold = copiesSold;
    }

    public int getCopiesSold ()
    {
        return copiesSold;
    }
}

Query query = pm.newQuery (Magazine.class);
query.setResult ("price, copiesSold");
query.setResultClass (SalesData.class);
List results = (List) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processSalesData ((SalesData) itr.next ());
query.close (results);

```

The example above is simple enough; the names of the projected fields are matched to the setter methods in the result class. But what if the names don't match? What if the result expressions contain aggregates, mathematical expressions, and relation traversals, all of which contain symbols that *can't* match a field or setter method name?

JDO provides the answer in the form of result expression *aliases*. An alias is a label assigned to a particular result expression for the purposes of matching that expression to fields or methods in the result class. To demonstrate this, let's modify our example above to populate each `SalesData` object not with the price and copies sold of each magazine, but with the average price and total copies sold of all magazines published by each company.

Example 11.21. Result Aliases

```

Query query = pm.newQuery (Magazine.class);
query.setResult ("avg(price) as price, sum(copiesSold) as copiesSold");
query.setResultClass (SalesData.class);
query.setGrouping ("publisher");
List results = (List) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processSalesData ((SalesData) itr.next ());
query.close (results);

```

Earlier in this chapter, we mentioned that the default result string for a query is `distinct this as C`, where `C` is the unqualified name of the candidate class. Now, finally, the meaning of this default string should be clear. But just to make it concrete, here is an example:

Example 11.22. Taking Advantage of the Default Result String

```

public class Wrapper
{
    private Magazine mag;

    public void setMagazine (Magazine mag)
    {
        this.mag = mag;
    }

    public Magazine getMagazine ()
    {
        return mag;
    }
}

Query query = pm.newQuery (Magazine.class);
query.setResultClass (Wrapper.class);
List results = (List) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processWrapper ((Wrapper) itr.next ());
query.close (results);

```

In this query on the `org.mag.Magazine` class, the default result string is `distinct this as Magazine`. Because our result class has a corresponding `setMagazine` method, the query can automatically populate each `Wrapper` with a matching magazine.

11.8.2. Generic Result Class

Whenever the specified result class does not contain a public field or setter method matching a particular result expression, the query looks for a method named `put` that takes two `Object` arguments. If found, the query invokes that method with the result expression or alias as the first argument, and its value as the second argument. This not only means that you can include a generic `put` method in your custom result classes, but that any `Map` implementation is suitable for use as a query result class.

Example 11.23. Populating a Map

```

Query query = pm.newQuery (Magazine.class);
query.setResult ("title.toUpperCase (), copiesSold");
query.setResultClass (HashMap.class);
List results = (List) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    HashMap map = (HashMap) itr.next ();
    processData (map.get ("title.toUpperCase ()"), map.get ("copiesSold"));
}
query.close (results);

```

11.9. Single-String JDOQL

It is often convenient to represent an entire JDOQL query as a single string. JDO defines a single-string query form that encompasses all parts of a query: the unique flag, result string, result class, filter, parameters, variables, imports, grouping, ordering, and range.

Note

Kodo uses the single-string format to implement JDOQL subquery support, covered in [Section 9.6.4, “JDOQL Subqueries” \[553\]](#) of the Reference Guide.

We present the grammar for single-string JDOQL queries below. Clauses in square brackets are optional. Clauses in angle brackets are required, and represent portions of the Query API.

```
select [unique] [<result>] [into <result class>]
  [from <candidate class> [exclude subclasses]]
  [where <filter>]
  [parameters <parameter declarations>]
  [variables <variable declarations>]
  [imports <import declarations>]
  [group by <grouping>]
  [order by <ordering>]
  [range <start>, <end>]
```

Each keyword above can appear in all lower case or all upper case.

The `PersistenceManager` has a factory method designed specifically for constructing a query from a JDOQL string:

```
public Query newQuery (String query);
```

Additionally, any of the `PersistenceManager`'s query factory methods that accept a JDOQL filter string will also accept the JDOQL single-string format. See [Section 8.11, “Query Factory” \[223\]](#).

In each code block below, we create and execute a standard JDOQL query, then demonstrate the same process using the single-string format. As you'll see, translating any query into single-string form is just a matter of plugging the corresponding strings into the grammar above.

```
Query query = pm.newQuery ();
query.setResult ("distinct title");
query.setClass (Magazine.class);
query.setFilter ("price < :p");
query.setRange (10, 20);
List results = (List) query.execute (new Double (10.0));

Query query = pm.newQuery (Magazine.class, "select distinct title "
+ "where price < :p range 10, 20");
List results = (List) query.execute (new Double (10.0));
```

```
Query query = pm.newQuery ();
query.setUnique (true);
query.setCandidates (pm.getExtent (Magazine.class, false));
query.setFilter ("title == 'JDJ'");
Magazine mag = (Magazine) query.execute ();

Query query = pm.newQuery ("select unique from org.mag.Magazine "
+ "exclude subclasses where title == 'JDJ'");
Magazine mag = (Magazine) query.execute ();
```

```
Query query = pm.newQuery ();
query.setResult ("publisher.name as pub, avg(price) as price");
query.setResultClass (org.mag.pub.PubPrice.class);
query.setClass (Magazine.class);
query.setGrouping ("publisher having avg(price) < :p");
query.setOrdering ("publisher.name ascending");
```

```

query.setRange (3, Long.MAX_VALUE);
List results = (List) query.execute (new Double (10.0));

Query query = pm.newQuery ("select publisher.name as pub, avg(price) as price "
+ "into org.mag.pub.PubPrice from org.mag.Magazine "
+ "group by publisher having avg(price) < :p "
+ "order by publisher.name ascending range 3, Long.MAX_VALUE");
List results = (List) query.execute (new Double (10.0));

```

On a final note, single-string queries are mutable. You can construct a single-string query as in the examples above, then change the result string, filter, or any other part of the query through the standard Query APIs. This is especially useful for temporary properties like the result range, and for setting the candidate class:

```

Query query = pm.newQuery (Magazine.class, "select title where "
+ "title.startsWith ('J')");
query.setRange (10, 20);
Magazine mag = (Magazine) query.execute ();

```

11.10. Named Queries

Named queries provide a means to define complex or commonly used queries in metadata. These queries have all the capabilities of queries created in code, including support for parameters, aggregates, and projections.

11.10.1. Defining Named Queries

You typically define named queries in *query metadata* files. These files use a `.jdoquery` extension, but otherwise follow the same naming and placement rules we outlined for `.jdo` files in [Section 5.7, “Metadata Placement” \[200\]](#).

Query metadata also has the same basic structure as persistence metadata. There is a root `jdoquery` element that contains package elements, which in turn contain class elements. `extensions` are sprinkled throughout. The only novel element is `query`.

```

<!ELEMENT jdoquery (extension*, (package|query)+, extension*)>
<!ELEMENT package (extension*, class+, extension*)>
<!ATTLIST package name CDATA #REQUIRED>

<!ELEMENT class (query+)>
<!ATTLIST class name CDATA #REQUIRED>

<!ELEMENT query ((#PCDATA|extension)*)>
<!ATTLIST query name CDATA #REQUIRED>
<!ATTLIST query language CDATA #IMPLIED>

<!ELEMENT extension ANY>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>

```

The `query` element defines a named query. You can place `query` elements just below the document's root element or at the class level. Queries within the root element do not have an automatic candidate class. Queries in a `class` element default to that candidate class.

Named queries are expressed in single-string form, which we detailed in the previous section. The text of the query goes within the `query` element. This element also has the following attributes:

- `name`: The name of the query. This attribute is required.
- `language`: The language of the query text. Defaults to `javax.jdo.query.JDOQL`. In [Chapter 17, SQL Queries \[320\]](#) we discuss how to write SQL queries, and vendors may offer support for additional query languages.

The following example defines both a root-level and a class-level named query.

Example 11.24. Query Metadata Document

The `salesReport` query generates sales information. The `findByTitle` query, on the other hand, finds a `Magazine` by title. Note that this query does not use a `from` clause to name its candidate class; its candidate class defaults to the surrounding class element.

```
<?xml version="1.0"?>
<jdoquery>
  <query name="salesReport">
    select title, price * copiesSold from org.mag.Magazine
  </query>
  <package name="org.mag">
    <class name="Magazine">
      <query name="findByTitle">select unique where title == :t</query>
    </class>
  </package>
</jdoquery>
```

The `Magazine` query metadata file above might be in any of the following locations:

- `org/mag/Magazine.jdoquery`
- `org/mag/package.jdoquery`
- `org/package.jdoquery`
- `package.jdoquery`

You do not have to use separate `.jdoquery` files to define named queries, however. You can integrate query definitions right into your persistence metadata (`.jdo` files) or mapping metadata (`.orm` files). Just add `query` elements below the root element, or after the last `field` and before any `fetch-group` elements in any class. Here is what it looks like to define the queries above in `Magazine`'s persistence metadata, rather than in a `.jdoquery` file:

Example 11.25. Persistence Metadata Document

```
<?xml version="1.0"?>
<jdo>
  <query name="salesReport">
    select title, price * copiesSold from org.mag.Magazine
  </query>
  <package name="org.mag">
    <class name="Magazine" objectid-class="Magazine$ObjectId">
      <field name="isbn" primary-key="true"/>
      <field name="title" primary-key="true"/>
      <field name="articles">
        <collection element-type="Article" dependent-element="true"/>
      </field>
      <query name="findByTitle">select unique where title == :t</query>
      <fetch-group name="detail">
        <field name="publisher" fetch-depth="0"/>
        <field name="articles" fetch-depth="0"/>
      </fetch-group>
    </class>
  </package>
</jdo>
```

```

        </fetch-group>
    </class>
</package>
</jdo>

```

11.10.2. Executing Named Queries

```
public Query newNamedQuery (Class cls, String name);
```

At runtime, you obtain named queries with the `PersistenceManager.newNamedQuery` method. The `Class` argument specifies the class that defines the query. Use `null` if the query is defined outside the scope of a class. The `String` argument is the name of the query.

Before returning a named query, the system populates it with the information provided in the metadata, then compiles it to make sure it is valid. You can still change the query before executing it, using the methods of the **Query** interface. These changes won't affect other query instances returned from subsequent calls to `newNamedQuery`.

The example below executes the two queries we defined in **Example 11.24, “Query Metadata Document”** [251].

Example 11.26. Executing Named Queries

```

PersistenceManager pm = ...;
Query query = pm.newNamedQuery (null, "salesReport");
List sales = (List) query.execute ();
for (Iterator itr = sales.iterator (); itr.hasNext ());
{
    Object[] salesData = (Object[]) itr.next ();
    processSalesData ((String) salesData[0], (Double) salesData[1]);
}
query.close (sales);

query = pm.newNamedQuery (Magazine.class, "findByTitle");
Magazine jdj = (Magazine) query.execute ("JDJ");

```

11.11. Delete By Query

Queries are useful not only for finding objects, but for efficiently deleting them as well. For example, you might delete all records created before a certain date. Rather than bring these objects into memory and delete them individually, JDO allows you to perform a single bulk delete based on JDOQL criteria.

Warning

This feature is only partially implemented in this Kodo release. In particular, note that objects deleted by query are not properly cleared from the **datastore cache** and that certain event callbacks may not be specification-compliant. Until this feature is fully implemented, we recommend isolating classes subject to delete by query in a separate cache which can be evicted manually or through timeouts.

Delete by query is accomplished through the following `Query` methods:

```
public long deletePersistentAll ();
public long deletePersistentAll (Object[] parameters);
public long deletePersistentAll (Map parameters);
```

These methods work similarly to the `Query.execute` methods. Instead of returning matching objects, however, the `deleteAll` methods delete the matching instances from the database, returning the number of objects deleted.

The following example deletes all subscriptions whose expiration date has passed.

Example 11.27. Delete By Query

```
Query query = pm.newQuery (Subscription.class, "expirationDate < :today");
long deleted = query.deletePersistentAll (new Object[] { new Date () });
```

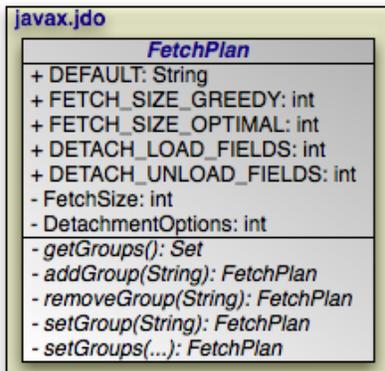
11.12. Conclusion

The `Query` interface is a powerful weapon in your JDO arsenal. In this chapter, we focused on how to use queries in conjunction with the datastore-neutral, object-oriented JDO query language, JDOQL. In **Chapter 17, *SQL Queries*** [320], we detail how to partner the `Query` interface with SQL instead.

Chapter 12. FetchPlan

The previous chapter showed you how to retrieve objects with JDO queries. But how do you control which fields and relations of those objects are fetched immediately, and which are loaded lazily? Similarly, when you use the `PersistenceManager` to lookup an object by JDO identity, or when you traverse a persistent relation, how does the JDO implementation decide which fields to populate? The answer lies in fetch groups.

Section 5.4, “Field Element” [196] introduced the default fetch group. This is the metadata-defined set of fields that load by default whenever you retrieve an object. **Section 5.5, “Fetch Group Element”** [198] showed you how to define additional custom fetch groups for additional control. By specifying the set of active fetch groups at runtime, you control the persistent fields that are populated immediately, versus those that are populated lazily. The interface you use to specify the set of active fetch groups is the `FetchPlan`.



`PersistenceManager`, `Query`, and `Extent` instances expose their fetch plans through their respective `getFetchPlan` methods.

```
public FetchPlan getFetchPlan ();
```

The `PersistenceManager`'s plan dictates the set of active fetch groups used when loading objects by JDO identity, or when traversing persistent relations. When a new query is created through the `PersistenceManager`'s `newQuery` or `newNamedQuery` methods, the query inherits the fetch plan settings from the `PersistenceManager`.

As you expect, `Query` and `Extent` plans apply to the objects loaded through the execution of the query or iteration of the extent. You can modify a `Query` or `Extent`'s fetch plan independently of the owning `PersistenceManager`'s plan; changes to one do not affect the other.

Note

Kodo extends the standard `FetchPlan` interface with the `KodoFetchPlan` interface and its relational sub-interface, the `JDBCFetchPlan`. **Section 9.3.7, “KodoFetchPlan”** [542] of the Reference Guide details these interfaces.

```
public static final String DEFAULT = "default";
public Collection getGroups ();
public FetchPlan addGroup (String group);
public FetchPlan removeGroup (String group);
public FetchPlan setGroup (String group);
public FetchPlan setGroups (Collection groups);
public FetchPlan setGroups (String[] groups);
```

FetchPlan defines several methods for manipulating the set of active fetch groups, and a single `getGroups` accessor to retrieve an immutable view of the set. Each mutator returns a reference to the `FetchPlan` instance for method chaining.

The `DEFAULT` constant represents the default fetch group.

```
public static int FETCH_SIZE_GREEDY = -1;
public static int FETCH_SIZE_OPTIMAL = 0;
public int getFetchSize ();
public FetchPlan setFetchSize (int size);
```

The fetch size is a hint to the implementation on how to process result sets. `FETCH_SIZE_GREEDY` eagerly loads all result objects. `FETCH_SIZE_OPTIMAL` leaves the result set processing strategy up to the JDO implementation. And any positive number `N` instructs the JDO runtime to process results as you access them in batches of `N`.

Note

Kodo excels at large result set handling. Under the proper settings, Kodo can even process result sets consisting of more objects than can fit into JVM memory.

Example 12.1. Using the FetchPlan

```
PersistenceManager pm = ...;
Query query = pm.newQuery (Magazine.class, "publisher.name == 'Random House'");
query.getFetchPlan ().setFetchSize (50).addGroup ("search-results");
List mags = (List) query.execute ();
for (Iterator itr = mags.iterator (); itr.hasNext ();)
    processMagazine ((Magazine) itr.next ());
```

12.1. Detachment Options

In addition to controlling the active fetch groups, the fetch plan allows you to control the `PersistenceManager`'s detach behavior. We covered detachment in [Section 8.7, “Detach and Attach Functionality” \[219\]](#).

```
public static int DETACH_LOAD_FIELDS;
public static int DETACH_UNLOAD_FIELDS;
public int getDetachmentOptions ();
public void setDetachmentOptions (int flags);
```

The bit flags above control what state is available in your detached objects. By default (`flags = 0`), the set of fields available in a detached instance is determined is the set of fields that were loaded when the original object was detached. The `DETACH_LOAD_FIELDS` flag instructs the JDO implementation to load the fields in your active fetch groups before detaching, guaranteeing that these fields will be available in the detached instance. The `DETACH_UNLOAD_FIELDS` flag, on the other hand, ensures that fields outside of your active fetch groups will **not** be available in the detached instance, even if they were loaded in the original managed object. You can combine these flags using bitwise operators.

Example 12.2. Using Detachment Options

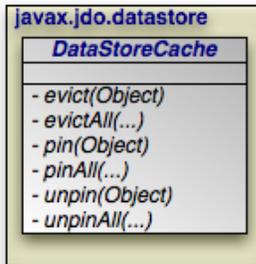
```
PersistenceManager pm = ...;
pm.getFetchPlan ().setDetachmentOptions (FetchPlan.DETACH_LOAD_FIELDS
| FetchPlan.DETACH_UNLOAD_FIELDS);
```

Note

Kodo's extended `kodo.jdo.KodoFetchPlan` also includes a `DETACH_ALL_FIELDS` flag.

Kodo currently does not support the `DETACH_LOAD_FIELDS` flag unless you also set the `DETACH_UNLOAD_FIELDS` flag, as in the example above.

Chapter 13. DataStoreCache



Recall that each `PersistenceManager` maintains a cache of persistent objects. Though this cache may improve performance, it is primarily in place to maintain data integrity and to satisfy JDO's **uniqueness requirement**. Thus, many vendors also implement a level 2 datastore cache for added performance. JDO's `DataStoreCache` interface gives you a standard way to access your vendor's level 2 cache.

```
public DataStoreCache getDataStoreCache ();
```

Datastore caches act across `PersistenceManagers`, at the `PersistenceManagerFactory` level. Some vendors may also go beyond the factory level to per-JVM and clustered caches. The `PersistenceManagerFactory`'s `getDataStoreCache` method retrieves the datastore cache. Vendors who do not support a level 2 data cache will implement all `DataStoreCache` operations as no-ops.

Note

Kodo extends the standard `DataStoreCache` interface with the `KodoDataStoreCache` interface to provide additional functionality. You can cast any `DataStoreCache` in Kodo to a `KodoDataStoreCache`.

As [Section 10.1, “Data Cache” \[560\]](#) in the Reference Guide explains, Kodo allows you to define multiple named caches. The `DataStoreCache` facade in Kodo delegates to any of these caches as necessary. The `KodoPersistenceManagerFactory` also supplies a method to retrieve a `DataStoreCache` facade to a specific named cache. Interacting with a specific named cache can be more efficient than forcing Kodo to calculate which cache to delegate to for each operation on the generic facade.

```
public void evict (Object oid);
public void evictAll ();
public void evictAll (Object[] oids);
public void evictAll (Collection oids);
```

The `evict` methods tell the cache to release data. Each method takes one or more JDO identity objects, and releases the cached data for the corresponding persistent instances. The `evictAll` method with no arguments clears the cache. Eviction is useful when the datastore is changed by a separate process outside your JDO implementation's control. In this scenario, you typically have to manually evict the data from the datastore cache; otherwise the JDO runtime, oblivious to the changes, will maintain its stale copy.

Note

In addition to manual eviction, Kodo offers per-class data timeouts and cron-style eviction schedules for caches. See [Section 10.1, “Data Cache” \[560\]](#) in the Reference Guide for details.

```
public void pin (Object oid);
public void pinAll (Object[] oids);
public void pinAll (Collection oids);
public void unpin (Object oid);
public void unpinAll (Object[] oids);
public void unpinAll (Collection oids);
```

Most caches are of limited size. Pinning a JDO identity object to the cache ensures that the cache will not expire the data for the corresponding instance, unless you manually evict it. Note that even after manual eviction, the data will get pinned again the next time it is fetched from the store. You can only remove a pin and make the data once again available for normal cache overflow eviction through the `unpin` methods. Use pinning when you want a guarantee that a certain object will always be available from cache, rather than requiring a datastore trip.

Example 13.1. Using the DataStoreCache

```
/**
 * This callback is invoked after an outside process runs to update the given
 * Subscription records. It evicts the records from the cache so that the
 * JDO application does not receive stale data.
 */
public void postBatchProcess (PersistenceManagerFactory pmf, long[] subIds)
{
    DataStoreCache cache = pmf.getDataStoreCache ();
    for (int i = 0; i < subIds.length; i++)
        cache.evict (new LongIdentity (Subscription.class, subIds[i]));
}
```

Chapter 14. JDOR

Most enterprise data today is stored in *relational databases*. Based on the relational data model developed in 1970 by E. F. Codd, relational databases store data in a collection of tables. Columns in each table represent the data's attributes, and rows represent individual records. Keys link records together, both within the same table and across tables. The Structured Query Language (SQL) operates on the stored data.

JDO / Relational, or *JDOR* for short, is a subset of the JDO specification dedicated to relational databases. JDOR standardizes how to map objects to relational tables, how to perform queries in SQL rather than JDOQL, and how to interact with database sequences. The remainder of this document covers each of these topics in detail.

Chapter 15. Mapping Metadata

Object-relational mapping is the process of mapping persistent classes to relational database tables. In JDOR, you perform object-relational mapping through *mapping metadata*. Mapping metadata uses XML to describe how to link your object model to your relational model. With a few notable exceptions, every persistent class and every persistent field must have mapping metadata.

Note

Kodo offers tools to automate mapping. See [Chapter 7, Mapping \[489\]](#) in the Reference Guide.

15.1. Mapping Metadata Placement

By default, JDOR implementations expect to find mapping metadata integrated into the **persistence metadata** defined in your `.jdo` files. You can, however, instruct the implementation to look for separate mapping files by supplying a value for the `javax.jdo.option.Mapping` key in the `Properties` used to construct your `PersistenceManagerFactory` (see [Section 6.3, “PersistenceManagerFactory Construction” \[205\]](#)). When this key is set, JDOR completely ignores any mapping metadata in your `.jdo` files. Instead, the implementation looks for mapping metadata in separate `.orm` files. The placement rules for these `.orm` files are very similar to the rules for `.jdo` files: the mapping document for a persistent class must be available as a resource from the class' class loader, and must exist in one of two standard locations:

1. In a resource called `<class-name>-<mapping>.orm`, where `<class-name>` is the unqualified name of the class the document applies to, and `<mapping>` is the value of the `javax.jdo.option.Mapping` property. The resource must be located in the same package as the class.
2. In a resource called `package-<mapping>.orm`, where `<mapping>` is the value of the `javax.jdo.option.Mapping` property. The resource should be placed in the corresponding package, or in any ancestor package. Package-level documents should contain the mappings for all the persistence-capable classes in the package, except those classes that have individual `<class-name>-<mapping>.orm` resources associated with them. They may also contain the mappings for classes in any sub-packages.

Assuming you are using a standard Java class loader, these rules imply that for a `javax.jdo.option.Mapping` setting of `hsqldb` and a class `Magazine` defined by the file `org/mag/Magazine.class`, the corresponding mapping document could be defined in any of the following files:

- `org/mag/Magazine-hsqldb.orm`
- `org/mag/package-hsqldb.orm`
- `org/package-hsqldb.orm`
- `package-hsqldb.orm`

Because mapping documents are loaded as resources, JDOR implementations can also read them from `jar` files.

Note

Kodo offers additional options for mapping metadata placement, including the ability to define mappings yourself at runtime. See [Section 7.5, “Mapping Factory” \[503\]](#)

15.2. Mapping Metadata DTD

Below we present the Document Type Definition for `.orm` files. Other than the root `orm` element, the same elements and attributes apply when integrating mapping metadata into `.jdo` files. When using integrated mapping, simply add the mapping attributes and elements below to your existing class and field metadata.

```

<!ELEMENT orm (extension*, package+, extension*)>
<!ELEMENT package (extension*, (class|sequence)+, extension*)>
<!ATTLIST package name CDATA ''>
<!ELEMENT sequence (extension*)>
<!ATTLIST sequence name CDATA #REQUIRED>
<!ATTLIST sequence strategy (nontransactional|transactional|contiguous) #IMPLIED>
<!ATTLIST sequence datastore-sequence CDATA #IMPLIED>
<!ATTLIST sequence factory-class CDATA #IMPLIED>
<!ELEMENT class (extension*, datastore-identity?, inheritance?, version?, join*, field*, extension*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class table CDATA #IMPLIED>
<!ELEMENT datastore-identity (extension*, column*, extension*)>
<!ATTLIST datastore-identity strategy CDATA 'native'>
<!ATTLIST datastore-identity sequence CDATA #IMPLIED>
<!ATTLIST datastore-identity column CDATA #IMPLIED>
<!ELEMENT inheritance (extension*, join?, discriminator?, extension*)>
<!ATTLIST inheritance strategy CDATA #IMPLIED>
<!ELEMENT discriminator (extension*, column*, index?, extension*)>
<!ATTLIST discriminator strategy CDATA #IMPLIED>
<!ATTLIST discriminator column CDATA #IMPLIED>
<!ATTLIST discriminator indexed (true|false|unique) #IMPLIED>
<!ATTLIST discriminator value CDATA #IMPLIED>
<!ELEMENT version (extension*, column*, index?, extension*)>
<!ATTLIST version strategy CDATA #IMPLIED>
<!ATTLIST version column CDATA #IMPLIED>
<!ATTLIST version indexed (true|false|unique) #IMPLIED>
<!ELEMENT join (extension*, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST join outer (true|false) 'false'>
<!ATTLIST join table CDATA #IMPLIED>
<!ATTLIST join column CDATA #IMPLIED>
<!ATTLIST join delete-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST join indexed (true|false|unique) #IMPLIED>
<!ATTLIST join unique (true|false) #IMPLIED>
<!ELEMENT column (extension*)>
<!ATTLIST column name CDATA #IMPLIED>
<!ATTLIST column jdbc-type CDATA #IMPLIED>
<!ATTLIST column sql-type CDATA #IMPLIED>
<!ATTLIST column length CDATA #IMPLIED>
<!ATTLIST column scale CDATA #IMPLIED>
<!ATTLIST column allows-null (true|false) #IMPLIED>
<!ATTLIST column target CDATA #IMPLIED>
<!ATTLIST column target-field CDATA #IMPLIED>
<!ATTLIST column default-value CDATA #IMPLIED>
<!ELEMENT field (extension*, join?, embedded?, element?, key?, value?, order?,
  column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field value-strategy CDATA #IMPLIED>
<!ATTLIST field sequence CDATA #IMPLIED>
<!ATTLIST field serialized (true|false) #IMPLIED>
<!ATTLIST field table CDATA #IMPLIED>
<!ATTLIST field column CDATA #IMPLIED>
<!ATTLIST field delete-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST field indexed (true|false|unique) #IMPLIED>
<!ATTLIST field unique (true|false) #IMPLIED>
<!ATTLIST field mapped-by CDATA #IMPLIED>
<!ELEMENT embedded (extension*, field*, extension*)>
<!ATTLIST embedded null-indicator-column CDATA #IMPLIED>
<!ELEMENT element (extension*, embedded?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST element serialized (true|false) #IMPLIED>
<!ATTLIST element table CDATA #IMPLIED>
<!ATTLIST element column CDATA #IMPLIED>
<!ATTLIST element delete-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST element indexed (true|false|unique) #IMPLIED>
<!ATTLIST element unique (true|false) #IMPLIED>
<!ELEMENT key (extension*, embedded?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST key mapped-by CDATA #IMPLIED>
<!ATTLIST key serialized (true|false) #IMPLIED>
<!ATTLIST key table CDATA #IMPLIED>
<!ATTLIST key column CDATA #IMPLIED>
<!ATTLIST key delete-action (restrict|cascade|null|default|none) #IMPLIED>

```

```

<!ATTLIST key indexed (true|false|unique) #IMPLIED>
<!ATTLIST key unique (true|false) #IMPLIED>

<!ELEMENT value (extension*, embedded?, column*, foreign-key?, index?, unique?, extension*)>
<!ATTLIST value serialized (true|false) #IMPLIED>
<!ATTLIST value table CDATA #IMPLIED>
<!ATTLIST value column CDATA #IMPLIED>
<!ATTLIST value delete-action (restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST value indexed (true|false|unique) #IMPLIED>
<!ATTLIST value unique (true|false) #IMPLIED>

<!ELEMENT order (extension*, column*, index?, extension*)>
<!ATTLIST order column CDATA #IMPLIED>
<!ATTLIST order indexed (true|false|unique) #IMPLIED>

<!ELEMENT index (extension*)>
<!ATTLIST index name CDATA #IMPLIED>
<!ATTLIST index table CDATA #IMPLIED>
<!ATTLIST index unique (true|false) 'false'>

<!ELEMENT foreign-key (extension*)>
<!ATTLIST foreign-key name CDATA #IMPLIED>
<!ATTLIST foreign-key table CDATA #IMPLIED>
<!ATTLIST foreign-key delete-action (restrict|cascade|null|default) 'restrict'>
<!ATTLIST foreign-key update-action (restrict|cascade|null|default) 'restrict'>
<!ATTLIST foreign-key deferred (true|false) #IMPLIED>

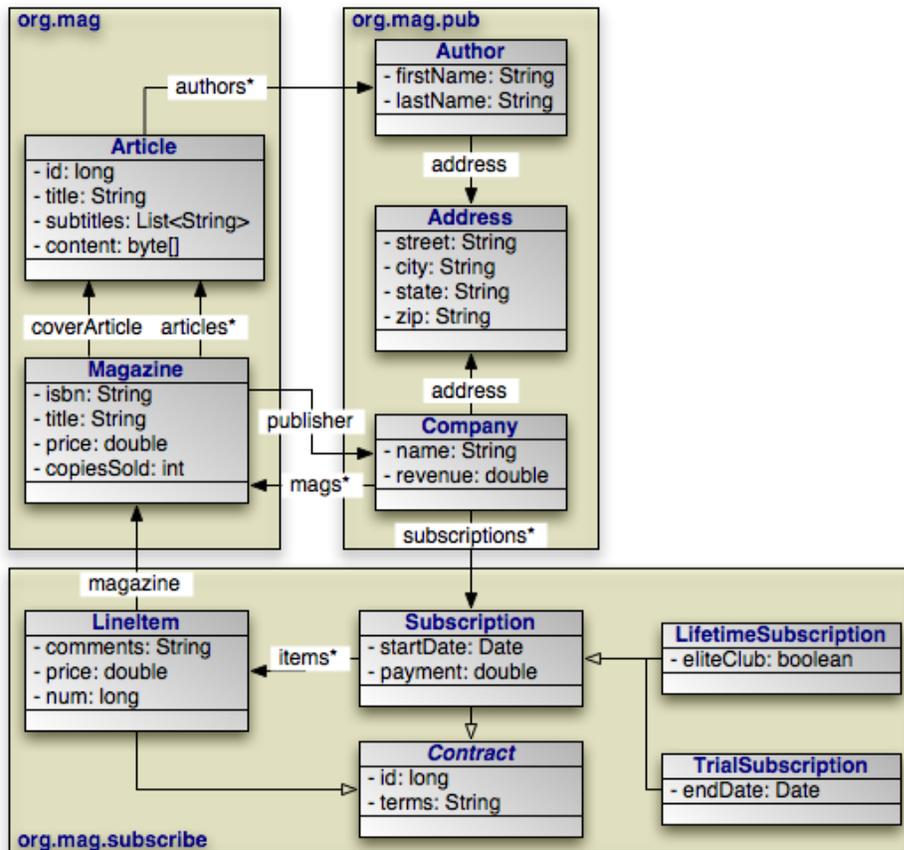
<!ELEMENT unique (extension*)>
<!ATTLIST unique name CDATA #IMPLIED>
<!ATTLIST unique table CDATA #IMPLIED>
<!ATTLIST unique deferred (true|false) 'false'>

<!ELEMENT extension ANY>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>

```

Mapping metadata's basic structure is virtually identical to that of persistence metadata, as defined in **Chapter 5, *Metadata* [193]**. The root `orm` element contains `package` elements, which contain `class` elements, which contain `fields`. The content of each mapping element, however, is very different than the content of its persistence metadata equivalent. The following sections discuss this content in detail.

Throughout this chapter, we will draw on the object model introduced in **Chapter 5, *Metadata* [193]**. We present that model again below. As we discuss various aspects of mapping metadata, we will zoom in on specific areas of the model and show how we map the object layer to the relational layer.



15.3. Sequences

Sequences are generators that are commonly used to create identity values or populate persistent fields. The package-level `sequence` element defines a named sequence. It has no child elements other than possible `extensions`, but uses the following attributes:

- **name:** The unqualified name of the sequence. Classes in the same package as the sequence can use this name as-is; classes outside of this package must refer to the sequence by its fully-qualified name, which is the sequence's defining package name + "." + the sequence name. This attribute is required.
- **strategy:** The sequence strategy. If not given, the implementation chooses a suitable default strategy. The strategy must be one of:
 - **nontransactional:** The sequence is updated outside of the current transaction. If the transaction rolls back, the sequence does not roll back.
 - **transactional:** The sequence is updated within the current transaction. If the current transaction rolls back, the sequence also rolls back. Transactional behavior is only guaranteed for datastore transactions. All sequences may behave like nontransactional sequences during optimistic transactions.
 - **contiguous:** Contiguous sequences guarantee that consecutive values will be monotonically increasing. All contiguous sequences are transactional.
- **datastore-sequence:** Names the native database sequence from which to extract values. This attribute is optional.

- `factory-class`: The fully qualified name of a sequence factory class. This class must have a static no-arg `newInstance` method that returns a `Sequence` instance. The `Sequence` interface is covered in detail in [Chapter 16, *Sequence*](#) [318] This attribute is optional.

Note

Kodo allows you to use a **plugin string** describing one of Kodo's built-in sequence implementations for the `factory-class` attribute. See the Reference Guide's [Section 9.7, “Generators”](#) [553] for built-in sequence options.

Example 15.1. Named Sequences

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <sequence name="ArticleSeq" datastore-sequence="ART_SEQ" />
    ...
  </package>
  <package name="org.mag.pub">
    <sequence name="AuthorSeq" factory-class="Author$SequenceFactory" />
    ...
  </package>
  <package name="org.mag.subscribe">
    <sequence name="ContractSeq" strategy="transactional" />
    ...
  </package>
</orm>
```

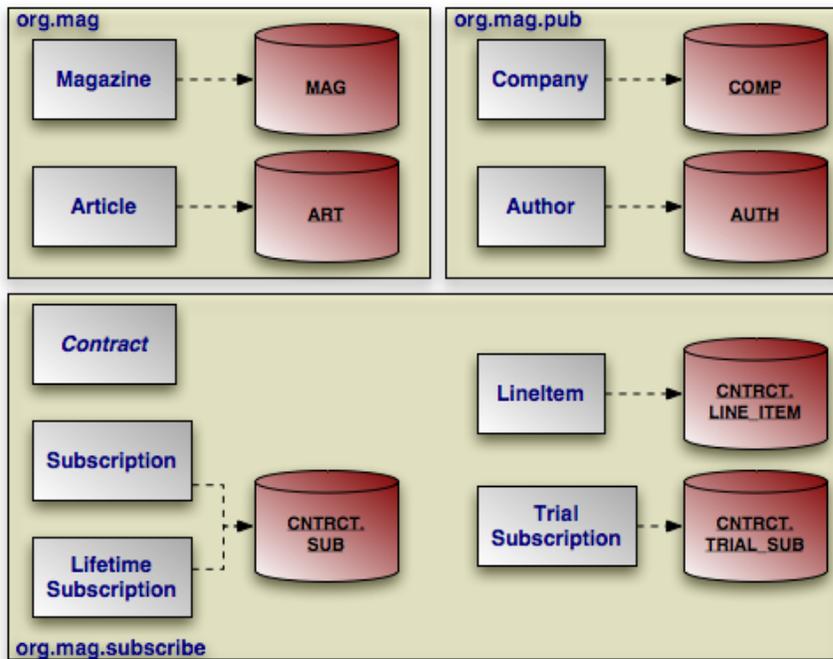
15.4. Class Table

As in persistence metadata, mapping metadata uses the `class` element and its `name` attribute to identify persistent classes. Mapping metadata, however, expands the `class` element with the `table` attribute.

The `table` attribute specifies the name of the table whose rows represent instances of this class. If the table is in a non-default schema, you can specify this attribute as `<schema-name>.<table-name>`. This attribute is optional, because not all classes must be mapped to unique tables. We explore the mapping of classes to tables in more depth in [Section 15.8, “Inheritance”](#) [273]

Sometimes, some of the fields in a class are mapped to secondary tables. In that case, use the class' `table` attribute to name what you consider the class' primary table. Later, we will see how to map certain fields to other tables.

The example below maps classes to tables according to the following diagram. Note that the abstract `Contract` class is left unmapped. The `SUB`, `TRIAL_SUB`, and `LINE_ITEM` tables are in the `CNTRCT` schema; all other tables are in the default schema.



Note that the example does not include our model's `Address` class in the mapping metadata document. In [Chapter 5, Metadata \[193\]](#), we defined `Address` as an *embedded only* class. That means that `Address` instances are only stored as part of other records; therefore `Address` does not have its own mapping.

Example 15.2. Mapping Classes

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <sequence name="ArticleSeq" datastore-sequence="ART_SEQ"/>
    <class name="Magazine" table="MAG">
      ...
    </class>
    <class name="Article" table="ART">
      ...
    </class>
  </package>
  <package name="org.mag.pub">
    <sequence name="AuthorSeq" factory-class="Author$SequenceFactory"/>
    <class name="Company" table="COMP">
      ...
    </class>
    <class name="Author" table="AUTH">
      ...
    </class>
  </package>
  <package name="org.mag.subscribe">
    <sequence name="ContractSeq" strategy="transactional"/>
    <class name="Contract">
      ...
    </class>
    <class name="Subscription" table="CNTRCT.SUB">
      ...
    </class>
    <class name="LifetimeSubscription">
      ...
    </class>
    <class name="TrialSubscription" table="CNTRCT.TRIAL_SUB">
      ...
    </class>
    <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
      ...
    </class>
  </package>
</orm>
```

15.5. Datastore Identity

In a relational database, a table's *primary key* is the set of columns whose values uniquely identify a row within that table. JDOR requires all persistent class tables to have primary keys, though these keys may be logical (i.e. there is no requirement that the database enforce the uniqueness of the primary key column values).

Under JDO's **application identity** type, the primary key field mappings are enough to infer the primary key columns of each class' table. Classes that use **datastore identity**, on the other hand, must describe their tables' primary key using the `datastore-identity` mapping element. This element is the first child of the `class` element, and has the following attributes:

- `strategy`: A strategy for auto-generating the primary key value. JDOR recognizes the strategies below, and vendors may define additional proprietary strategies.
 - `native`: Leave primary key value generation to the JDOR implementation. This is the default strategy if none is specified.
 - `sequence`: Use a sequence to generate the primary key value. This strategy is used in conjunction with the `sequence` attribute below.
 - `autoassign`: The database automatically assigns a primary key value on insert. This might be accomplished with an auto-increment column or with database triggers.

When you use the `autoassign` strategy, calling `JDOHelper.getObjectId` or `PersistenceManager.getObjectId` with a persistent-new object may cause the `PersistenceManager` to flush. The `PersistenceManager` will insert the new object and retrieve the database-assigned primary key value so that it can return the new instance's proper persistent identity.

- `identity`: The primary key column is managed by the database as an identity type.
- `increment`: The JDOR implementation will select the highest current primary key value, and increment it by one to get the primary key value for the next inserted object.
- `uuid-string`: Generates a 128-bit UUID unique within the network and represents the result as a 16-character string. For more information on UUIDs, see the IETF UUID draft specification at: <http://www1.ics.uci.edu/~ejw/authoring/uuid-guid/>
- `uuid-hex`: Same as `uuid-string`, but represents the UUID as a 32-character hexadecimal string.

Not all standard datastore identity strategies will be implemented by all vendors. For example, there is no way for a vendor that uses numeric datastore identity values to support the `uuid-string` and `uuid-hex` strategies.

Note

Kodo supports the `native`, `sequence`, `autoassign`, and `identity` datastore identity strategies. Kodo treats `identity` and `autoassign` equivalently, as they are the same for most databases. Before using the `identity` or `autoassign` strategies, see [Section 5.3.3, “Autoassign / Identity Strategy Caveats” \[458\]](#)

Though Kodo does not define any custom datastore identity strategies, you can take over the generation of datastore identity values by using the `sequence` strategy in conjunction with a sequence that has a custom `factory-class`.

- `sequence`: Names a sequence to use to generate the primary key value. When this attribute is specified, the `strategy` attribute described above automatically defaults to a value of `sequence`.

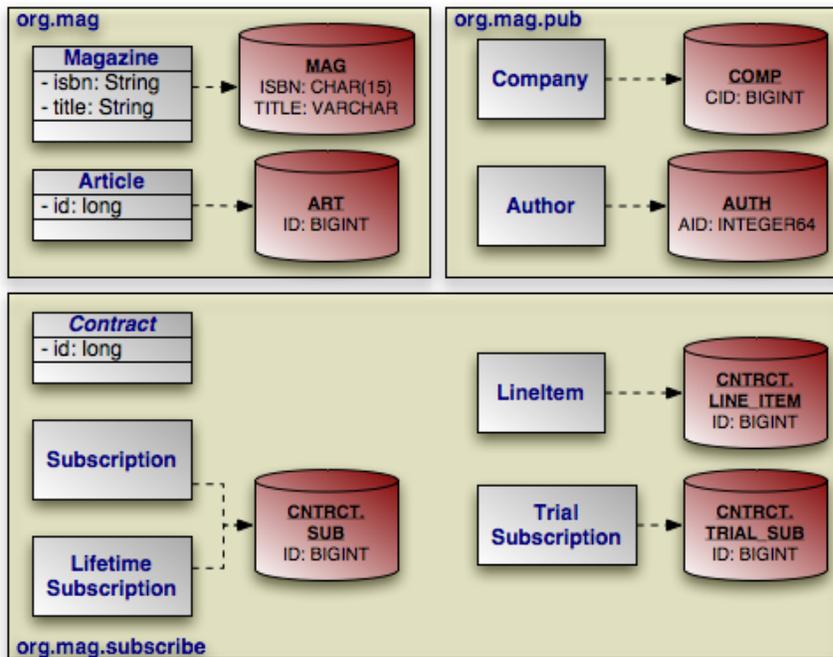
Note

If you specify `system` as the sequence attribute value, Kodo will use the system sequence defined by the `kodo.Sequence` configuration property instead of looking for a named sequence.

- `column`: The name of the primary key column.

In place of the `column` attribute above, you can embed `column` elements within the `datastore-identity` element. This allows you to define multiple datastore identity columns for implementations that require it, and to specify additional attributes aside from the column name. [Section 15.6, “Column” \[268\]](#) discusses the `column` element in detail.

The diagram below now includes primary key columns for our model's tables. The primary key column for `LineItem` uses non-standard type `INTEGER64`, and the `Magazine.isbn` field is mapped to a `CHAR(15)` column instead of a `VARCHAR(255)` column, which is the default for string fields. We do not need to point out either one of these oddities to the JDOR implementation for runtime use. If, however, we want to use the JDOR implementation to create our tables for us, it needs to know about any desired non-default column types. Therefore, the example following the diagram includes this data in its encoding of our mappings. The example also includes the mappings of primary key fields in our application identity classes; we will get to field mapping in [Section 15.11, “Field Mapping” \[287\]](#)



Example 15.3. Datastore Identity Mapping

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <sequence name="ArticleSeq" datastore-sequence="ART_SEQ"/>
    <class name="Magazine" table="MAG">
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      ...
    </class>
    <class name="Article" table="ART">
      <field name="id" column="ID"/>
      ...
    </class>
  </package>
  ...
  <package name="org.mag.pub">
    <class name="Company" table="COMP">
      <field name="cid" column="CID"/>
    </class>
    <class name="Author" table="AUTH">
      <field name="aid" column="AID"/>
    </class>
  </package>
  ...
  <package name="org.mag.subscribe">
    <class name="Contract" table="CNTRCT">
      <field name="id" column="ID"/>
    </class>
    <class name="Subscription" table="CNTRCT">
      <field name="id" column="ID"/>
    </class>
    <class name="Lifetime Subscription" table="CNTRCT">
      <field name="id" column="ID"/>
    </class>
    <class name="LineItem" table="CNTRCT">
      <field name="id" column="ID"/>
    </class>
    <class name="Trial Subscription" table="CNTRCT">
      <field name="id" column="ID"/>
    </class>
  </package>
</orm>
```

```

</package>
<package name="org.mag.pub">
  <sequence name="AuthorSeq" factory-class="Author$SequenceFactory"/>
  <class name="Company" table="COMP">
    <datastore-identity column="CID" strategy="autoassign"/>
    ...
  </class>
  <class name="Author" table="AUTH">
    <datastore-identity sequence="AuthorSeq">
      <column name="AID" sql-type="INTEGER64"/>
    </datastore-identity>
    ...
  </class>
</package>
<package name="org.mag.subscribe">
  <sequence name="ContractSeq" strategy="transactional"/>
  <class name="Contract">
    ...
  </class>
  <class name="Subscription" table="CNRCT.SUB">
    <field name="Contract.id" column="ID"/>
    ...
  </class>
  <class name="LifetimeSubscription">
    ...
  </class>
  <class name="TrialSubscription" table="CNRCT.TRIAL_SUB">
    ...
  </class>
  <class name="Subscription$LineItem" table="CNRCT.LINE_ITEM">
    <field name="Contract.id" column="ID"/>
    ...
  </class>
</package>
</orm>

```

15.6. Column

In the previous section, we saw that the `datastore-identity` element can take either a `column` attribute or nested `column` elements. This option is repeated on many elements throughout mapping metadata. The motivation for this dualistic approach is to make the common case easy, but retain the flexibility to handle more complex scenarios. When a mapping has a single column with default properties, you can use the `column` attribute to just name the column. When, on the other hand, a mapping has multiple columns or columns with non-default properties, you can fall back to nested `column` elements. We saw this in **Example 15.3, “Datastore Identity Mapping”** [267] above: most classes used the `column` attribute to simply name the primary key column, but `Author` and `Magazine` used a nested element in order to specify additional column information. Column elements have the following attributes:

- `name`: The column name. Later in this chapter, we will come across situations where a column is not in the expected table. In these cases, you can specify the column name as `<table-name>.<column-name>`, or even `<schema-name>.<table-name>.<column-name>`.

The column name is required for runtime use. The reason the `name` attribute isn't marked `#REQUIRED` in the DTD is that some vendors may allow you to partially-specify column information during the mapping process, and have the rest filled in by a vendor-supplied tool.

- `jdbc-type`: Determines what JDBC APIs the implementation uses to load and store data to the column. This attribute is only required when the column uses a non-default type for the associated data - for example, when a `String` is mapped to a `CLOB` column rather than a `CHAR` or `VARCHAR`.

Standard values for this attribute include all of the constant names from the `java.sql.Types` class, in either uppercase or lowercase: `BIGINT`, `bigint`, `BLOB`, `blob`, `DECIMAL`, `decimal`, `VARCHAR`, `varchar`, etc.

- `sql-type`: The database-specific column type name. This attribute is only used by vendors that support creating tables from your mapping metadata. During table creation, the vendor will use the text of the `sql-type` attribute as the declared column type. If no `sql-type` is given, the vendor will choose an appropriate default based on the column's JDBC type, length, and scale.

- `length`: The column length. This attribute is typically only used during table creation, though some vendors might use it to validate data before flushing. `CHAR` and `VARCHAR` columns typically default to a length of 255; other column types use the database default.
- `scale`: The number of decimal digits a numeric column can hold. This attribute is often used in conjunction with `length` to form the proper column type name during table creation.
- `allows-null`: Whether the column can store null values. Vendors may use this attribute both for table creation and at runtime; however, it is never required. Defaults to `false` for primary key columns and columns holding primitive field values, and `true` for all other columns.
- `default-value`: A database-assigned default value for the column if no value is given in the `INSERT SQL`. Vendors will pass this default on to the database during table creation. When a new object is being persisted with Java default field values, vendors may use this attribute to decide whether to include the columns for those fields in the generated `INSERT SQL`.

Note

Kodo uses a combination of a column's default value and JDO metadata's `null-value` field attribute to decide whether to include a column in an `INSERT` statement. The following table describes what value is inserted for a field with a default value, based on whether the column has a `default-value` and on the field's `null-value` attribute:

Table 15.1. Default Value Inserts

<code>default-value</code>	<code>null-value unset</code>	<code>null-value="none"</code>	<code>null-value="default"</code>
Set	Column not included in <code>INSERT</code>	<code>NULL</code> / Field value	Column not included in <code>INSERT</code>
Unset	<code>NULL</code> / Field value	<code>NULL</code> / Field value	Field type's Java default

- `target`: In joins, the column that this column joins to. We examine joins in detail in the next section.
- `target-field`: In joins, the primary key field of the related type that this column joins to. We examine joins in detail in the next section.

Note

Kodo also defines extensions to mark that the current mapping should not attempt to insert or update a column's value. See [Section 7.9.3, “Column Extensions” \[522\]](#) in the Reference Guide.

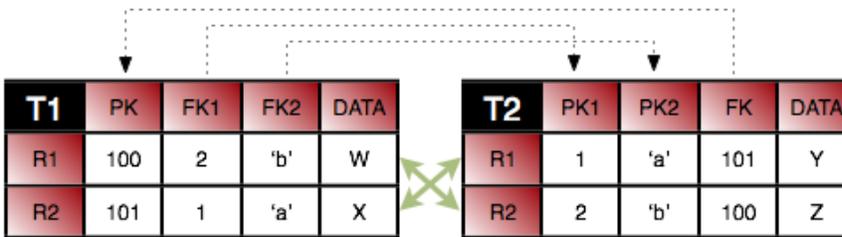
15.7. Joins

One of the mainstays of a relational database is the ability for one record to refer to another. This allows relational databases to join related records together to retrieve large chunks of data efficiently. A record A typically references a record B by setting some of its columns to the same values as B's primary key columns. The columns in A that mirror primary key columns in B are called *foreign key* columns. Most relational databases can enforce several special behaviors for foreign keys, which we explore in [Section 15.12, “Foreign Keys” \[309\]](#). For now, though, it is sufficient to remember that foreign keys are sets of columns that act as a logical reference to another record by storing that record's primary key values.

Note

In Kodo, logical foreign key columns do not have to link to primary key columns. You can link any columns. You can

also express joins in which some columns must have specific constant values. See the Reference Guide's [Section 7.6, “Non-Standard Joins”](#) [506] for details on Kodo's support for non-standard joins.



Consider the two tables above. Table T1 has a primary key column PK, two foreign key columns FK1 and FK2, and a standard data column DATA. Table T2 has two primary key columns PK1 and PK2, foreign key column FK, and a data column. The dotted arrows indicate that T1's foreign key columns link to T2's primary key columns, and T2's foreign key column links to T1's primary key column. A foreign key's table is called the *source* table, and the table it links to is called the *target* table.

By matching the data in the foreign key and primary key columns, you can see that row R1 in T1 references row R2 in T2 and vice versa. Conversely, row R2 in T1 references row R1 in T2 and vice versa. The solid arrows depict these cross-referencing relationships.

In JDOR, you specify a foreign key column's target primary key column with the column element's `target` attribute. So, assuming we're mapping something in a class using the T1 table, the link from T1 to T2 becomes:

```
<class name="..." table="T1">
  <...>
  <column name="FK1" target="PK1"/>
  <column name="FK2" target="PK2"/>
  </...>
</class>
```

Note that we did not qualify the column names above with the containing table name. The foreign key column names are not qualified because the foreign key columns are in the parent class' table. The target primary key column names are not qualified because JDOR defaults the target table based on the context of the mapping. For example, when you map a relation field, JDOR assumes the target table is the table of the related class.

A standard *forward* foreign key like the one above maps the *target* table records *referenced by* the current source table record. You can, however, also map an *inverse* foreign key. An inverse foreign key maps all the *source* table records *referencing* the current target table record. Inverse foreign keys only apply to mappings from the target table - in this case, T2. Thus, you must qualify the foreign key column names with their source table name:

```
<class name="..." table="T2">
  <...>
  <column name="T1.FK1" target="PK1"/>
  <column name="T1.FK2" target="PK2"/>
  <.../>
</class>
```

15.7.1. Shortcuts

Now let's shift our focus to the link from T2 to T1 through the T2 . FK foreign key column. This is easy enough to represent in mapping metadata:

```
<class name="..." table="T2">
```

```
<...>
  <column name="FK" target="PK"/>
<.../>
</class>
```

We can, however, simplify this even further. When the target table's primary key consists of a single column, you can omit the `target` attribute. JDOR assumes that the target is the lone primary key column of the target table (which is itself defaulted based on the mapping context):

```
<class name="..." table="T2">
  <...>
  <column name="FK"/>
  <.../>
</class>
```

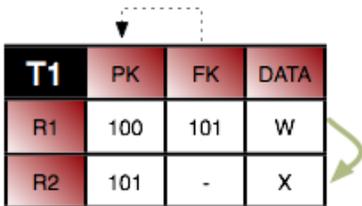
Recall from the previous section that when you only need to specify a column name, you can use the enclosing element's `column` attribute in place of a nested `column` element. This allows us to be even more concise:

```
<class name="..." table="T2">
  <... column="FK"/>
</class>
```

JDOR provides one other significant shortcut not seen here. When the foreign key column name is the same as the target primary key column name, you do not have to set the `target` attribute.

15.7.2. Self Joins

Foreign key columns do not have to reference records in other tables. Often, a table will use foreign keys to itself.



As you can see, R1 links to R2 through the FK foreign key column. The mapping is straightforward:

```
<class name="..." table="T1">
  <...>
  <column name="FK" target="PK"/>
  <.../>
</class>
```

Or, after applying available shortcuts:

```
<class name="..." table="T1">
  <... column="FK"/>
</class>
```

You may be wondering how to represent an *inverse* self-join. Because the source and target tables are the same, the naïve approach results in a mapping that looks identical to a forward join, which is obviously incorrect:

```
<class name="..." table="T1">
  <...>
  <column name="FK" target="PK"/>
  <.../>
</class>
```

The answer is simple and surprisingly consistent. When we created an inverse foreign key mapping between T1 and T2 above, we had to fully qualify the foreign key column names. To represent an inverse self join, just use the same trick:

```
<class name="..." table="T1">
  <...>
  <column name="T1.FK" target="PK"/>
  <.../>
</class>
```

Though the foreign key column is in the parent class' table and wouldn't normally require qualification, using the qualified name serves as a hint to the JDOR implementation that this is an inverse join.

15.7.3. Target Fields

JDOR allows you to specify a target primary key field name in place of a target column with the `target-field` attribute. This is simply an object-oriented shortcut for targeting the named field's column. For example:

```
<class name="A" table="T1">
  <field name="relationToB">
    <column name="FK1" target-field="id1"/>
    <column name="FK2" target-field="id2"/>
  </field>
</class>
<class name="B" table="T2">
  <field name="id1" column="PK1"/>
  <field name="id2" column="PK2"/>
</class>
```

Is equivalent to:

```
<class name="A" table="T1">
  <field name="relationToB">
    <column name="FK1" target="PK1"/>
    <column name="FK2" target="PK2"/>
  </field>
</class>
<class name="B" table="T2">
  <field name="id1" column="PK1"/>
  <field name="id2" column="PK2"/>
</class>
```

You can fully-qualify a target field name with its class, just as you can fully-qualify a target column name with its table.

Note

Target fields are useful when you rely on Kodo's mapping tools to create the schema, and you want to supply mapping hints without having to name the target columns. The Reference Guide Details Kodo's mapping tools in [Section 7.1, “Forward Mapping” \[489\]](#)

Also, Kodo does not require that target fields be primary key fields. See [Section 7.6, “Non-Standard Joins” \[506\]](#) in the Reference Guide.

15.8. Inheritance

In the 1990's programmers coined the term *impedance mismatch* to describe the difficulties in bridging the object and relational worlds. Perhaps no feature of object modeling highlights the impedance mismatch better than inheritance. There is no natural, efficient way to represent an inheritance relationship in a relational database.

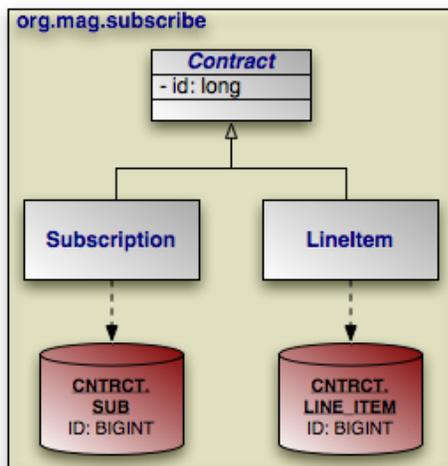
Luckily, JDOR provides very flexible object-relational mapping of inheritance trees, making the best of a bad situation. Each class in an inheritance hierarchy can use the `inheritance` element to describe its inheritance mapping. This element has a single attribute: `strategy`. In the list that follows, we examine JDOR's standard inheritance strategies. Some vendors may define additional proprietary strategies.

Note

Kodo does not define any non-standard inheritance models, but does allow you to create your own custom mappings. See the Reference Guide's [Section 7.10, “Custom Mappings” \[522\]](#) for details on writing custom inheritance strategies and plugin them in to Kodo.

15.8.1. subclass-table

The `subclass-table` inheritance strategy indicates that the current class is not mapped to any table at all, and that all of its fields must be mapped by subclasses into their own tables. This strategy is typically used with abstract base classes that are not represented in the relational schema. Classes that declare an inheritance strategy of `subclass-table` should not define the `class` element's `table` attribute, nor should they attempt to define mappings for any fields.



In our model, the abstract `Contract` class uses the `subclass-table` inheritance strategy. `Contract`'s two direct subclasses, `Subscription` and `LineItem`, each map all of `Contract`'s fields into their own tables.

Example 15.4. subclass-table Mapping

```
<class name="Contract">
  <inheritance strategy="subclass-table"/>
  ...
</class>
```

Note

The pattern of mapping the fields of an abstract, unmapped base class into the tables of each subclass is often referred to as a *horizontal* or *distributed* inheritance mapping.

15.8.1.1. Advantages

An advantage of using the `subclass-table` strategy for abstract base classes is that properties common to multiple persistent subclasses can be defined in the superclass without having to suffer the performance consequences and relational design restrictions inherent in other strategies (which we will examine shortly). Persisting and modifying instances of subclasses is efficient, typically only requiring a single `INSERT` or `UPDATE` statement. Loading relations to these subclasses is also efficient.

15.8.1.2. Disadvantages

Though relations to mapped subclasses of a `subclass-table` class are very efficient, relations to the unmapped base class itself are equally *inefficient*. When the concrete subclass is not known, the related object could be in any of the subclass tables, making joins through the relation impossible. Subclasses can even use repeated primary key values, forcing the JDOR implementation to record more than just the related primary key values in the database.

This ambiguity also affects queries and identity lookups: queries against a `subclass-table` base class require either multiple `SELECT` statements (one for each mapped subclass), or a complex `SQL UNION`.

Note

Kodo provides metadata extensions you can use to indicate that a field declared as a relation to a base class is actually a relation to a specific subclass. These extensions often alleviate the need for mapping a relation to a `subclass-table` type. See [Section 6.4, “Metadata Extensions” \[483\]](#)

When a relation to a `subclass-table` class cannot be avoided, Kodo stores either the related object's primary key values or its stringified object id. If the type uses datastore identity or has a concrete application identity class, Kodo stores the primary key values. If the type uses application identity and has an abstract identity class, Kodo must resort to storing the stringified identity object.

15.8.1.3. Additional Considerations

Here are some additional caveats to consider when using the `subclass-table` inheritance mapping:

- *Declaring classes abstract.* You are not required to make all `subclass-table` classes abstract. However, we recommend that you do so, as any attempt to persist a `subclass-table` base class instance will result in an exception on flush.
- *Application identity.* When a `subclass-table` superclass uses application identity, you must observe one of the following two restrictions:
 1. The primary key values in each of the tables that extend the `subclass-table` superclass must be unique. In our model, that means that there can be no row in `CNTRCT.SUB` with the same primary key value as a row in `CNTRCT.LINE_ITEM`. This is because a call to `getObjectById` with an application identity instance cannot identi-

fy which subclass the ID is associated with.

2. Rather than having a single application identity class associated with the entire class hierarchy, each subclass declares its own application identity class. The inheritance hierarchy of the application identity classes must exactly match the inheritance hierarchy of the persistent classes they represent, as discussed in **Section 4.5.2.1, “Application Identity Hierarchies”** [190].

15.8.2. new-table

The `new-table` inheritance strategy employs a new table to hold the fields of the class. You must specify the table name in the `class` element's `table` attribute.

`new-table` is the default strategy for base persistent classes and for subclasses of `subclass-table` classes. Classes that extend `non-subclass-table` base classes can also use this strategy to map their fields to a new table, rather than to the superclass table. The subclass table might contain only subclass state, or might re-map all superclass state as well.

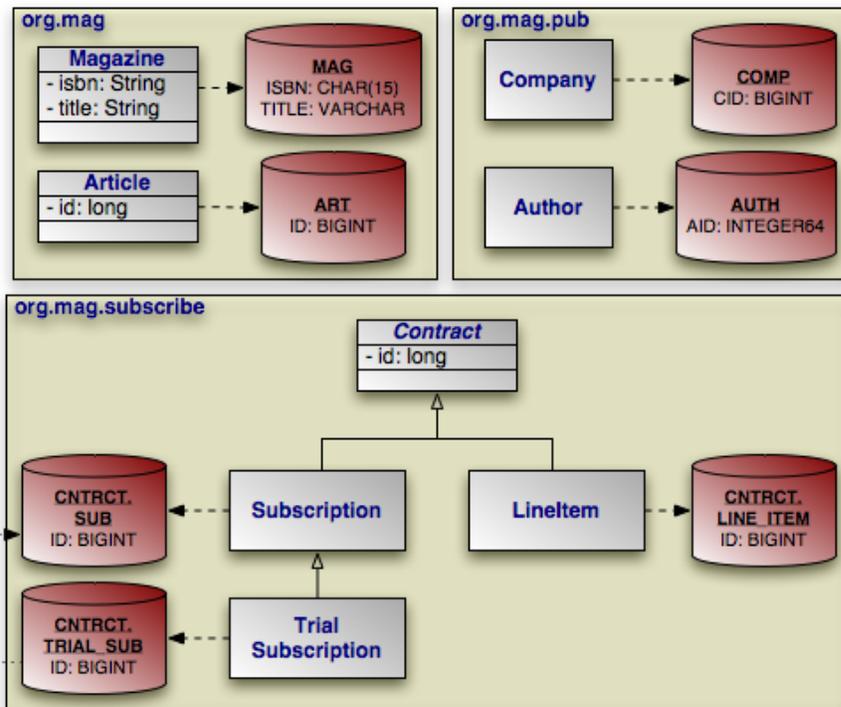
15.8.2.1. Joined

When the subclass table only contains subclass state, the JDOR implementation must be able to link corresponding records in the superclass and subclass tables together to retrieve all of the persistent state for a subclass instance. You tell the JDOR implementation how to do this by nesting a `join` element in your `inheritance` element. Thus, this is often referred to as a *joined* inheritance strategy.

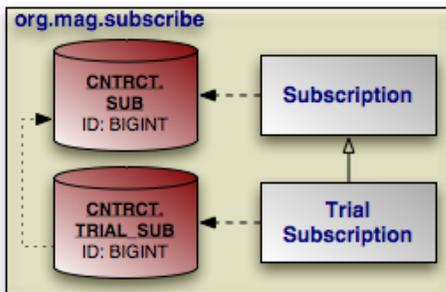
As its name implies, the `join` element maps a logical foreign key from the subclass table to the superclass table. It has a `column` attribute and nested `column` elements for representing the subclass table's foreign key columns. **Section 15.7, “Joins”** [269] demonstrated how to use `columns` to map joins. Typically, the subclass table's foreign key columns are also that table's primary key columns.

Note

Using joined subclass tables is also called *vertical* inheritance mapping.



All of the classes in our model except `Contract` and `LifetimeSubscription` (not pictured) use the `new-table` strategy. Most of these classes are either base classes or direct subclasses of `Contract`, a `subclass-table` class. In these cases, the `new-table` strategy is the default. `TrialSubscription`, however, extends `Subscription`, which is itself mapped to a table. Therefore, `TrialSubscription` could have mapped its fields to `Subscription`'s table.



Instead, `TrialSubscription` maps its declared fields to the `CNTRCT.TRIAL_SUB` table, and joins to the `CNTRCT.SUB` table for base class state. The join is made by linking the `CNTRCT.TRIAL_SUB.ID` foreign key column to the `CNTRCT.SUB.ID` primary key column. The example below shows how to represent this in mapping metadata.

Example 15.5. Joined Subclass Tables

```
<class name="TrialSubscription" table="CNTRCT.TRIAL_SUB">
  <inheritance strategy="new-table">
    <join>
      <column name="ID" target="CNTRCT.SUB.ID"/>
    </join>
  </inheritance>
  ...
</class>
```

That is the long version, however. JDOR is smart enough to default the target table to the superclass table, and we can apply **join shortcuts** to yield a much more concise representation:

```
<class name="TrialSubscription" table="CNTRCT.TRIAL_SUB">
  <inheritance strategy="new-table">
    <join column="ID"/>
  </inheritance>
  ...
</class>
```

15.8.2.1.1. Advantages

If you want to map a class to a table and none of the superclasses of that class are themselves mapped to a table (or the class has no persistent superclasses), then the `new-table` strategy is your only choice. It is only meaningful to discuss the advantages and disadvantages of the `new-table` strategy for classes that have a superclass mapped to a different table. For example, our model's `TrialSubscription` class extends `Subscription`, yet maps its declared fields to a different table. We could have mapped `TrialSubscription`'s fields to `Subscription`'s table; what made us choose to use a separate joined table instead?

1. Using joined subclass tables results in the most *normalized* database schema, meaning the schema with the least spurious or redundant data.
2. As more subclasses are added to the data model over time, the only schema modification that needs to be made is the addition of corresponding subclass tables in the database, rather than having to change the structure of existing tables.
3. Relations to a base class using the `new-table` strategy can be loaded through standard joins and can use standard foreign keys, as opposed to the machinations required to load relations to `subclass-table` base types.

15.8.2.1.2. Disadvantages

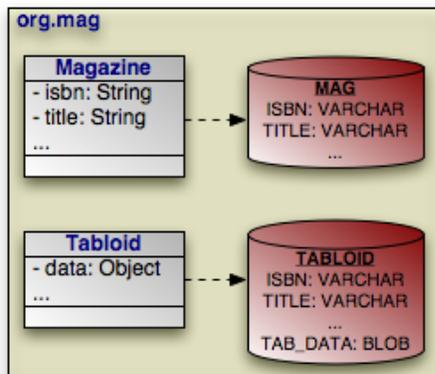
Using multiple joined tables slows things down. Retrieving any subclass requires one or more database joins, and storing subclasses requires multiple `INSERT` or `UPDATE` statements. This is only the case when persistence operations are performed on subclasses; if most operations are performed on the least-derived persistent superclass, then this mapping is very fast.

Note

When executing a select against a hierarchy that uses joined subclass table inheritance, you must consider how to load subclass state. [Section 5.7, “Eager Fetching” \[473\]](#) in the Reference Guide describes Kodo's options for efficient data loading.

15.8.2.2. Table Per Class

Like the joined `new-table` strategy, the *table-per-class* `new-table` strategy maps a subclass to its own table. Unlike the joined strategy, however, the subclass table includes all state for an instance of the corresponding class. Thus to load a subclass instance, the JDOR implementation must only read from the subclass table; it does not need to join to superclass tables.



Suppose that our sample model's `Magazine` class has a subclass `Tabloid`. The classes are mapped using the table-per-class strategy, as in the diagram above. In a table-per-class mapping, `Magazine`'s table `MAG` contains all state declared in the base `Magazine` class. `Tabloid` maps to a separate table, `TABLOID`. This table contains not only the state declared in the `Tabloid` subclass, but all the base class state from `Magazine` as well. Thus the `TABLOID` table would contain columns for `isbn`, `title`, and other `Magazine` fields.

Example 15.6. Table Per Class Mapping

```
<class name="Tabloid" table="TABLOID">
  <inheritance strategy="new-table"/>
  <field name="Magazine.isbn" column="ISBN"/>
  <field name="Magazine.title" column="TITLE"/>
  ...
  <field name="data" column="TAB_DATA"/>
</class>
```

Notice that only the lack of a `join` within the `inheritance` element differentiates a table-per-class strategy from a joined strategy. Also, notice that a table-per-class subclass explicitly re-maps all of its inherited fields into its own table.

15.8.2.2.1. Advantages

The table-per-class strategy is very efficient when operating on instances of a known class. Under these conditions, the strategy never requires joining to superclass or subclass tables. Reads, joins, inserts, updates, and deletes are all efficient in the absence of polymorphic behavior. Also, as in the joined strategy, adding additional classes to the hierarchy does not require modifying existing class tables.

15.8.2.2.2. Disadvantages

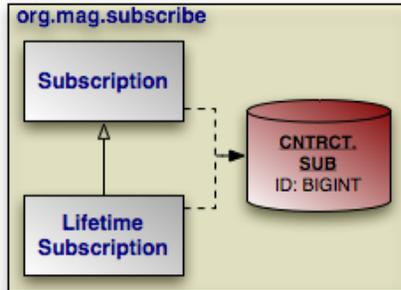
Polymorphic relations to a non-leaf classes in a table-per-class hierarchy have many limitations. In some ways, they are similar to relations to a `subclass-table` base class. When the concrete subclass is not known, the related object could be in any of the subclass tables, making joins through the relation impossible. This ambiguity also affects identity lookups and queries; these operations require multiple SQL `SELECT`s (one for each possible subclass), or a complex `UNION`.

Note

Section 7.8.1, “Table Per Class” [519] in the Reference Guide describes the limitations Kodo places on table-per-class mapping.

15.8.3. superclass-table

`superclass-table` is the default strategy for subclasses of `new-table` and other `superclass-table` classes. In this strategy, the subclass' fields are mapped to superclass' table. Classes that use the `superclass-table` inheritance strategy should not specify the `class` element's `table` attribute.



In our model, `Subscription` is mapped to the `CNTRCT.SUB` table. `LifetimeSubscription`, which extends `Subscription`, adds its field data to this table as well.

Example 15.7. superclass-table Mapping

```

<class name="LifetimeSubscription">
  <inheritance strategy="superclass-table" />
  ...
</class>
  
```

Note

Mapping subclass state to the superclass table is often called *flat* inheritance mapping.

15.8.3.1. Advantages

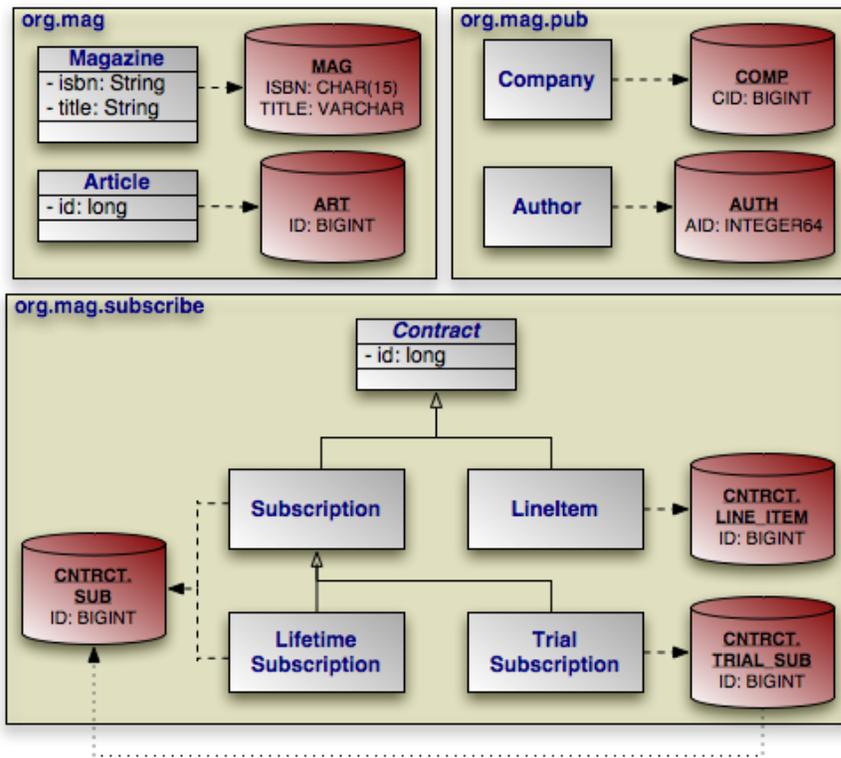
`superclass-table` inheritance mapping is the fastest of all inheritance models, since it never requires a join to retrieve a persistent instance from the database. Similarly, persisting or updating a persistent instance requires only a single `INSERT` or `UPDATE` statement. Finally, relations to any class within a `superclass-table` inheritance hierarchy are just as efficient as relations to a base class.

15.8.3.2. Disadvantages

The larger the inheritance model gets, the "wider" the mapped table gets, in that for every field in the entire inheritance hierarchy, a column must exist in the mapped table. This may have undesirable consequence on the database size, since a wide or deep inheritance hierarchy will result in tables with many mostly-empty columns.

15.8.4. Putting it All Together

Now that we have covered JDOR's inheritance strategies, we can update our mapping document with inheritance information. Here is the complete model:



And here is the corresponding mapping metadata:

Example 15.8. Inheritance Mapping

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <sequence name="ArticleSeq" datastore-sequence="ART_SEQ"/>
    <class name="Magazine" table="MAG">
      <!-- not strictly necessary, since this is the default -->
      <inheritance strategy="new-table"/>
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      ...
    </class>
    <class name="Article" table="ART">
      <!-- not strictly necessary, since this is the default -->
      <inheritance strategy="new-table"/>
      <field name="id" column="ID"/>
      ...
    </class>
  </package>
  <package name="org.mag.pub">
    <sequence name="AuthorSeq" factory-class="Author$SequenceFactory"/>
    <class name="Company" table="COMP">
      <datastore-identity column="CID" strategy="autoassign"/>
      <!-- not strictly necessary, since this is the default -->
      <inheritance strategy="new-table"/>
      ...
    </class>
    <class name="Author" table="AUTH">
      <datastore-identity sequence="AuthorSeq">
        <column name="AID" sql-type="INTEGER64"/>
      </datastore-identity>
      <!-- not strictly necessary, since this is the default -->
      <inheritance strategy="new-table"/>
      ...
    </class>
  </package>
  <package name="org.mag.subscribe">
    <sequence name="ContractSeq" strategy="transactional"/>
```

```

<class name="Contract">
  <inheritance strategy="subclass-table"/>
  ...
</class>
<class name="Subscription" table="CNTRCT.SUB">
  <!-- not strictly necessary, since this is the default -->
  <inheritance strategy="new-table"/>
  <field name="Contract.id" column="ID"/>
  ...
</class>
<class name="LifetimeSubscription">
  <!-- not strictly necessary, since this is the default -->
  <inheritance strategy="superclass-table"/>
  ...
</class>
<class name="TrialSubscription" table="CNTRCT.TRIAL_SUB">
  <inheritance strategy="new-table">
    <join column="ID"/>
  </inheritance>
  ...
</class>
<class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
  <!-- default -->
  <inheritance strategy="new-table"/>
  <field name="Contract.id" column="ID"/>
  ...
</class>
</package>
</orm>

```

15.9. Discriminator

The `superclass-table` inheritance strategy results in a single table containing records for two or more different classes in an inheritance hierarchy. Similarly, using the `new-table` strategy to create joined subclass tables results in the superclass table holding records for superclass instances as well as for the superclass state of subclass instances. When selecting data, JDOR needs a way to differentiate a row representing an object of one class from a row representing an object of another. That is the job of the `discriminator` mapping element.

The `discriminator` element is nested within the `inheritance` element. The least-derived mapped class in an inheritance hierarchy defines the discriminator. In other words, the first `new-table` class in a hierarchy declares the discriminator strategy and column; `subclass-table` superclasses do not declare a discriminator at all, and all subclasses of the first `new-table` class can only declare a discriminator value. The `discriminator` element has the following attributes:

- `strategy`: The discriminator strategy. JDOR recognizes several standard strategies, which we detail in the following sections. Vendors may also define their own proprietary strategies.

Note

Kodo defines the `final` discriminator strategy for base persistent classes that are not declared `final`, but nevertheless will not have any persistent subclasses. Using the `final` strategy for these classes is slightly more efficient than the `none` strategy described below. Kodo also allows you to plug in your own discriminator implementations. The Reference Guide's [Section 7.10, “Custom Mappings” \[522\]](#) has details on creating custom discriminators.

- `value`: A discriminator value for this class. This attribute is used with the `value-map` discriminator strategy; see [Section 15.9.2, “value-map” \[282\]](#)
- `column`: The column that holds the discriminator value. As you will see, not all discriminator strategies require a column. As with the `datastore-identity` element, you can use a nested `column` element in lieu of the `column` attribute.

15.9.1. class-name

The `class-name` discriminator strategy writes the name of the class each row represents to the discriminator column, indicated by the `column` attribute/element. Aside from its ease-of-use, the main advantage of this strategy is that a good JDOR implementation can calculate all of the subclasses of a given persistent type solely by querying the discriminator column. This means that you do not have to point the implementation at a manually-maintained list of persistent classes. It can be bothersome to keep such a list properly synchronized with a rapidly-changing object model during development, or to deploy with a full list of classes in a J2EE environment where multiple sub-applications share a single server.

On the other hand, storing class names in the database couples your data to your Java object model. In many projects, the persistent data will last far longer than your object mode; perhaps even longer than the Java language itself! For these projects, consider the `value-map` strategy.

15.9.2. value-map

The `value-map` strategy is like the `class-name` strategy, except that a designated symbolic value is written to the discriminator column instead of the class name. This means the JDOR implementation cannot determine a type's subclasses by querying the discriminator column alone, but it has the advantage of keeping your database independent of your object model.

To use the `value-map` strategy, the least-derived mapped class declares the discriminator column, and all concrete classes in the hierarchy (least-derived class included) use the `discriminator` element's `value` attribute to assign themselves a symbolic value. This value will be inserted into the discriminator column for rows representing instances of the class. In fact, the discriminator strategy automatically defaults to `value-map` whenever the `value` attribute is defined; you do not need to specify the `strategy` attribute as well.

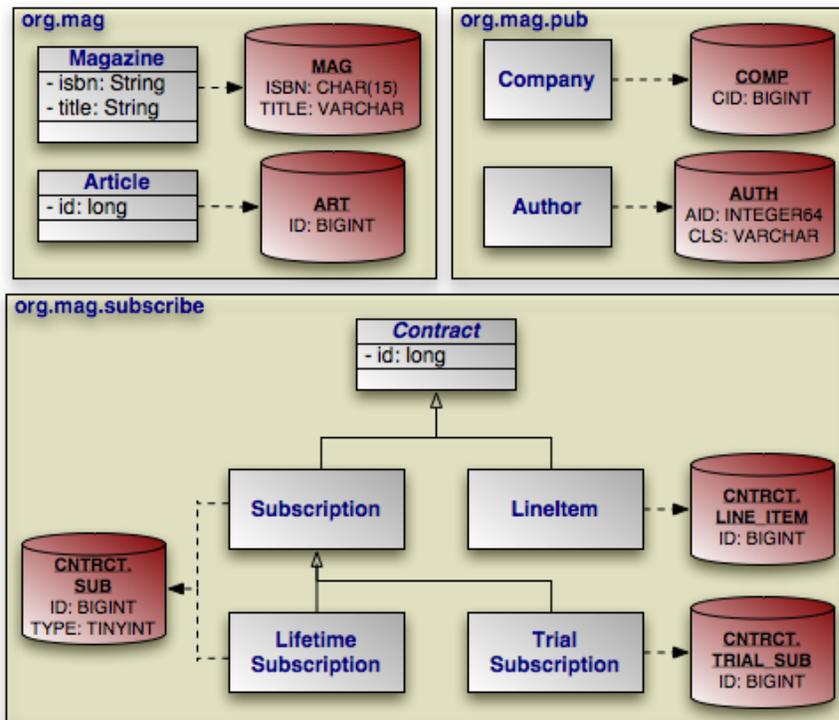
15.9.3. none

Setting the discriminator strategy to `none` indicates that there is no discriminator column. This is illegal for hierarchies using `subclass-table` inheritance, but is valid in other circumstances:

- Base classes that you will never extend do not need a discriminator.
- Subclasses of a `subclass-table` type do not need a discriminator, unless they themselves have additional subclasses.
- A *vertical* hierarchy that uses joined subclass tables via the `new-table` inheritance strategy does not require a discriminator column, *if your database supports outer joins*. An outer join is a join in which the related record does not have to exist. When reading an object, the JDOR implementation can outer join to all possible subclass tables and use the presence or absence of a matching row in each table to calculate the class of the instance. Obviously, this is not a very efficient strategy for large hierarchies; for these we recommend using a column-based strategy such as `value-map`.

15.9.4. Putting it All Together

We can now translate our newfound knowledge of JDOR discriminators into concrete JDOR mappings. We first extend our diagram with discriminator columns:



Next, we present the updated mapping document. There are several things to notice in this revised version:

1. We have removed explicit inheritance strategy declarations for classes using the default strategy.
2. Magazine, Article, and Company use the none discriminator strategy because we do not plan on subclassing them.
3. Author uses the class-name strategy; perhaps we plan on selling our application framework to other developers, and we want them to be able to add new subclasses of Author easily.
4. Contract is abstract and is not a mapped class (it uses subclass-table inheritance), and therefore it has no discriminator. Its two subclasses Subscription and Linetitem, however, are each the least-derived mapped classes in the hierarchy, and so they each declare a discriminator. Note that though they are sibling classes, they can each use a different discriminator mapping because their superclass is unmapped.
5. Subscription's discriminator strategy automatically defaults to value-map because we define the value attribute. Under the value-map strategy, we must also define a unique discriminator value for every concrete subclass of Subscription.
6. Discriminator columns default to type VARCHAR. Subscription and its subclasses use numeric discriminator values; therefore, when we map Subscription's discriminator column we use a nested column element rather than the column attribute. The nested element allows us to tell the JDOR implementation the proper jdbc-type of the column.

Example 15.9. Discriminator Mapping

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <sequence name="ArticleSeq" datastore-sequence="ART_SEQ"/>
    <class name="Magazine" table="MAG">
      <inheritance>
```

```

        <discriminator strategy="none"/>
    </inheritance>
    <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
    </field>
    <field name="title" column="TITLE"/>
    ...
</class>
<class name="Article" table="ART">
    <inheritance>
        <discriminator strategy="none"/>
    </inheritance>
    <field name="id" column="ID"/>
    ...
</class>
</package>
<package name="org.mag.pub">
    <sequence name="AuthorSeq" factory-class="Author$SequenceFactory"/>
    <class name="Company" table="COMP">
        <datastore-identity column="CID" strategy="autoassign"/>
        <inheritance>
            <discriminator strategy="none"/>
        </inheritance>
        ...
    </class>
    <class name="Author" table="AUTH">
        <datastore-identity sequence="AuthorSeq">
            <column name="AID" sql-type="INTEGER64"/>
        </datastore-identity>
        <inheritance>
            <discriminator column="CLS" strategy="class-name"/>
        </inheritance>
        ...
    </class>
</package>
<package name="org.mag.subscribe">
    <sequence name="ContractSeq" strategy="transactional"/>
    <class name="Contract">
        <inheritance strategy="subclass-table"/>
        ...
    </class>
    <class name="Subscription" table="CNTRCT.SUB">
        <inheritance>
            <discriminator value="1">
                <column name="TYPE" jdbc-type="tinyint"/>
            </discriminator>
        </inheritance>
        <field name="Contract.id" column="ID"/>
        ...
    </class>
    <class name="LifetimeSubscription">
        <inheritance>
            <discriminator value="2"/>
        </inheritance>
        ...
    </class>
    <class name="TrialSubscription" table="CNTRCT.TRIAL_SUB">
        <inheritance strategy="new-table">
            <join column="ID"/>
            <discriminator value="3"/>
        </inheritance>
        ...
    </class>
    <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
        <inheritance>
            <discriminator strategy="none"/>
        </inheritance>
        <field name="Contract.id" column="ID"/>
        ...
    </class>
</package>
</orm>

```

15.10. Version

Section 9.1, “Transaction Types” [226] introduced optimistic transactions. In order to prevent one optimistic transaction from blindly overwriting the changes made by a concurrent transaction, JDOR versions your objects. The `version` element dictates what form this versioning takes.

The `version` element appears just after the `inheritance` element in a mapping document. Only the least-derived mapped

classes in an inheritance hierarchy use the `version` element. The element has the following attributes:

- `strategy`: The version strategy. We review the standard JDO strategies in the following sections. Vendors may define additional non-standard strategies.

Note

Kodo does not define any non-standard version strategies, but does allow you to plug in custom strategies of your own. The Reference Guide's [Section 7.10, “Custom Mappings” \[522\]](#) documents version customization in detail.

Kodo also allows you to define multiple *lock groups* for fine-grained control over optimistic versioning. See [Section 5.8, “Lock Groups” \[476\]](#) in the Reference Guide for more information on lock groups.

- `column`: The column that holds the version value. Not all version strategies require a column.

As with other elements, the `column` attribute can be replaced with a nested `column` element.

15.10.1. none

As its name implies, the `none` version strategy performs no versioning at all. Using concurrent optimistic transactions under this strategy is dangerous.

15.10.2. version-number

The `version-number` strategy stores a monotonically increasing version number in the column indicated by the `column` attribute/element. When a dirty object's database record is being updated, the JDOR implementation checks its in-memory version number against the database version number. If the database version is higher, the implementation knows that another transaction has modified the object since the current `PersistenceManager` last read its state, and it aborts the transaction to preserve data integrity. If the version number is the same, on the other hand, the implementation increments the number and proceeds with the flush.

The `version-number` strategy is the fastest, most efficient, and most accurate version strategy. Its only drawback is that it requires a dedicated database column. If you are mapping to legacy tables you might not be able to use it.

15.10.3. date-time

The `date-time` strategy is exactly like the `version-number` strategy, but it timestamps each update rather than storing an increasing integer. This strategy is mainly present to support existing tables that use time-based versioning. We do not recommend it for new schemas, since it is theoretically possible for two updates to happen so close together that they have the same timestamp. The `version-number` strategy is a better choice. In addition to being slightly more efficient, it does not suffer from these sorts of timing failures.

15.10.4. state-comparison

The `state-comparison` version strategy does not require a database column. It works by comparing the last-read values of your object with the database on flush. If any values in the database don't match the recorded last-read value, then some other transaction must have concurrently modified the data, and the flush is aborted.

Unfortunately, the `state-image` strategy is much more memory-hungry than other strategies, because the JDOR runtime has to store the last-read values of all modified fields. It also results in more complex `UPDATE SQL` statements as the JDOR implementation verifies that no column has been changed by a concurrent transaction. Finally, it suffers from the following limitations:

- Only simple, exact field values can be used in state comparisons. If a commit only changes a `float` or `Collection` field,

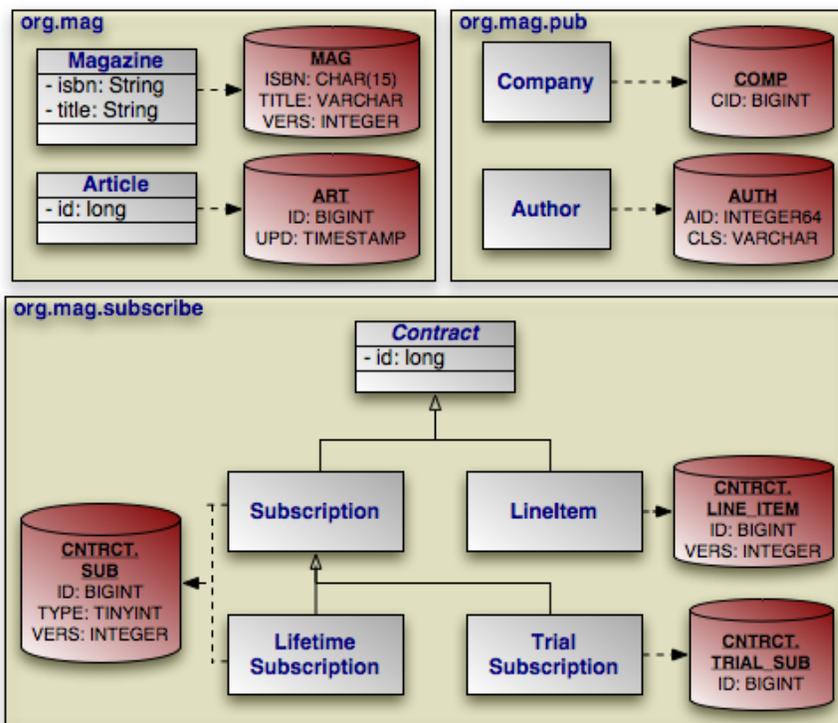
subsequent commits will not detect any difference in the object's version, because these fields are not compared. Thus it is possible for one transaction to unknowingly overwrite another.

- If two concurrent transactions make changes to fields that reside in a disjoint set of tables, the second transaction may overwrite the first. For example, if one transaction modifies a `TrialSubscription` instance by only changing fields mapped to `CNTRCT.SUB`, and a concurrent transaction modifies the same instance but only changes fields mapped to `CNTRCT.TRIAL_SUB`, one transaction might overwrite the other.

Due to these shortcomings, we only recommend using the `state-comparison` strategy when performing optimistic transactions on legacy tables without a version column.

15.10.5. Putting it All Together

Here is our model with version columns added:



And here is our updated mapping data. We made sure that all version strategies are represented for illustrative purposes. Note that as an unmapped subclass-table type, `Contract` does not declare a version. Its direct subclasses `Subscription` and `Linetitem`, however, each define version mappings. `TrialSubscription` does not declare an additional version on its `CNTRCT.TRIAL_SUB` table, because it joins down to its superclass table. We have also continued to consolidate our mappings; this document leaves out default inheritance and discriminator mappings.

Example 15.10. Version Mapping

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <sequence name="ArticleSeq" datastore-sequence="ART_SEQ"/>
    <class name="Magazine" table="MAG">
      <version column="VERS" strategy="version-number"/>
      <field name="isbn">
```

```

        <column name="ISBN" jdbc-type="char" length="15"/>
    </field>
    <field name="title" column="TITLE"/>
    ...
</class>
<class name="Article" table="ART">
    <version column="UPD" strategy="date-time"/>
    <field name="id" column="ID"/>
    ...
</class>
</package>
<package name="org.mag.pub">
    <sequence name="AuthorSeq" factory-class="Author$SequenceFactory"/>
    <class name="Company" table="COMP">
        <datastore-identity column="CID" strategy="autoassign"/>
        <version strategy="none"/>
        ...
    </class>
    <class name="Author" table="AUTH">
        <datastore-identity sequence="AuthorSeq">
            <column name="AID" sql-type="INTEGER64"/>
        </datastore-identity>
        <inheritance>
            <discriminator column="CLS" strategy="class-name"/>
        </inheritance>
        <version strategy="state-comparison"/>
        ...
    </class>
</package>
<package name="org.mag.subscribe">
    <sequence name="ContractSeq" strategy="transactional"/>
    <class name="Contract">
        <inheritance strategy="subclass-table"/>
        ...
    </class>
    <class name="Subscription" table="CNRCT.SUB">
        <inheritance>
            <discriminator value="1">
                <column name="TYPE" jdbc-type="tinyint"/>
            </discriminator>
        </inheritance>
        <version column="VERS" strategy="version-number"/>
        <field name="Contract.id" column="ID"/>
        ...
    </class>
    <class name="LifetimeSubscription">
        <inheritance>
            <discriminator value="2"/>
        </inheritance>
        ...
    </class>
    <class name="TrialSubscription" table="CNRCT.TRIAL_SUB">
        <inheritance strategy="new-table">
            <join column="ID"/>
            <discriminator value="3"/>
        </inheritance>
        ...
    </class>
    <class name="Subscription$LineItem" table="CNRCT.LINE_ITEM">
        <version column="VERS" strategy="version-number"/>
        <field name="Contract.id" column="ID"/>
        ...
    </class>
</package>
</orm>

```

15.11. Field Mapping

The following sections enumerate the myriad of field mappings JDOR supports. JDOR augments standard persistence metadata's `field` element with many new object-relational attributes and sub-elements. As we explore the library of standard mappings, we introduce each of these enhancements in context.

Note

Kodo allows you to create custom field mappings for unsupported field types or database schemas. See the Reference Guide's [Chapter 7, Mapping \[489\]](#) for complete coverage of Kodo JDO's mapping capabilities.

15.11.1. Superclass Fields

As you may have already noticed from the mapping document we have been constructing throughout this chapter, subclasses can map superclass fields by setting the `field` element's `name` attribute to `<superclass-name>.<field-name>`. For example, `Subscription` maps its `Contract` superclass' `id` field:

```
<class name="Subscription" table="CNTRCT.SUB">
  <field name="Contract.id" column="ID"/>
  ...
</class>
```

If `Contract` were in a different package than `Subscription`, we would have to fully qualify `Contract`'s class name:

```
<class name="Subscription" table="CNTRCT.SUB">
  <field name="org.mag.subscribe.Contract.id" column="ID"/>
  ...
</class>
```

As we tour the standard JDOR field mappings throughout this chapter, keep in mind that any of them can be applied to a superclass field just as easily as to a field in the current class.

15.11.2. Basic Mapping

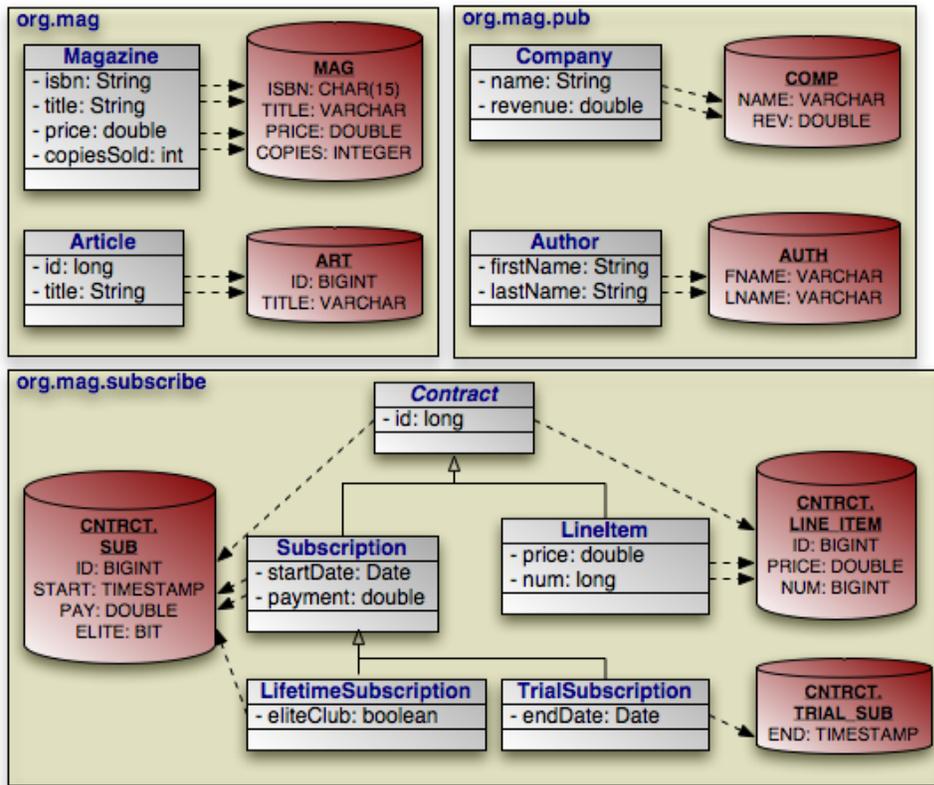
A *basic* field mapping stores the field value directly into a database column. Primitives, primitive wrappers, `Strings`, `Dates`, `Locales`, and any other supported type that can be easily translated into a native JDBC value default to basic mapping. In fact, you have already seen examples of basic field mappings in this chapter - for example, the mapping of `Magazine`'s primary key fields in [Example 15.3, “Datastore Identity Mapping” \[267\]](#).

To write a basic field mapping, just name the column that holds the field value. The `field` element accepts a `column` attribute or nested `column` elements for this purpose. We introduced columns and the attribute/element dichotomy already in [Section 15.6, “Column” \[268\]](#).

Note

Kodo stores Java 5 Enum values by storing the value name in any `CHAR` or `VARCHAR` column. Storing the name is safer than storing the ordinal position, in case you reorder your enum values or add new options at the beginning or middle of the value list.

Below we present an updated diagram of our model and its associated database schema, followed by its representation as a mapping metadata document. All basic fields are now mapped.



Example 15.11. Basic Field Mapping

```

<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      <field name="price" column="PRICE"/>
      <field name="copiesSold" column="COPIES"/>
      ...
    </class>
    <class name="Article" table="ART">
      <field name="id" column="ID"/>
      <field name="title" column="TITLE"/>
      ...
    </class>
  </package>
  <package name="org.mag.pub">
    <class name="Company" table="COMP">
      <field name="name" column="NAME"/>
      <field name="revenue" column="REV"/>
      ...
    </class>
    <class name="Author" table="AUTH">
      <field name="firstName" column="FNAME"/>
      <field name="lastName" column="LNAME"/>
      ...
    </class>
  </package>
  <package name="org.mag.subscribe">
    <class name="Subscription" table="CNTRCT.SUB">
      <field name="Contract.id" column="ID"/>
      <field name="startDate" column="START"/>
      <field name="payment" column="PAY"/>
      ...
    </class>
    <class name="LifetimeSubscription">
      <field name="eliteClub" column="ELITE"/>
    </class>
  </package>
  </orm>

```

```

    ...
  </class>
  <class name="TrialSubscription" table="CNTRCT.TRIAL_SUB">
    <field name="endDate" column="END"/>
    ...
  </class>
  <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
    <field name="Contract.id" column="ID"/>
    <field name="price" column="PRICE"/>
    <field name="num" column="NUM"/>
    ...
  </class>
</package>
</orm>

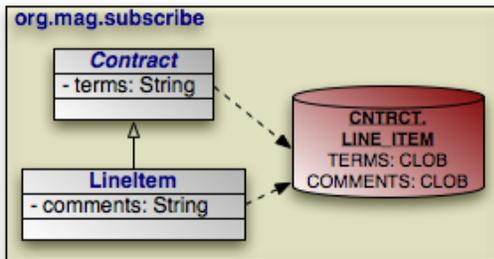
```

15.11.2.1. CLOB

Unlike Java, most relational databases cannot transparently store strings of arbitrary length. The standard database string types CHAR and VARCHAR are often limited to holding 255 characters or less. To store longer strings, you must use the CLOB column type.

CLOB stands for Character Large Object. In JDOR, mapping a Java String field to a database CLOB column is exactly like creating any other basic mapping. The only difference is that you must use the column element's jdbc-type attribute to tell the JDOR implementation to use JDBC's CLOB APIs rather than the standard string APIs.

Our sample model uses two large string fields: Contract.terms and LineItem.comments. Since Contract uses a subclass-table mapping, LineItem maps both these fields to its own table:



In mapping metadata, CLOB mapping looks like this:

Example 15.12. CLOB Field Mapping

```

<class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
  <field name="Contract.terms">
    <column name="TERMS" jdbc-type="clob"/>
  </field>
  <field name="comments">
    <column name="COMMENTS" jdbc-type="clob"/>
  </field>
  ...
</class>

```

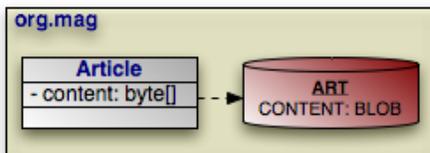
Note

In Kodo, you can set the column's `length` attribute to `-1` rather than setting its `jdbc-type` to `clob`. This approach allows Kodo to take advantage of some databases' ability to represent unlimited-length strings natively, without resorting to a CLOB. If your database does not support unlimited-length strings natively, Kodo falls back to CLOB handling.

15.11.2.2. BLOB

BLOBs are Binary Large Objects. JDOR uses BLOB columns to store byte arrays and serialized Java objects.

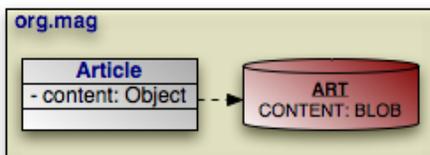
Mapping a `byte[]` field to a BLOB column is a basic mapping. For example, let's map our model's `Article.content` field to the `ART.CONTENT` BLOB column:



Example 15.13. Byte Array Field Mapping

```
<class name="Article" table="ART">
  <field name="content" column="CONTENT"/>
  ...
</class>
```

Mapping a serialized Object field to a BLOB column is like mapping a `byte[]` field, except that you must explicitly instruct JDOR to serialize the field value with the `serialized` attribute. Suppose that instead of a `byte[]`, our `Article.content` field is of type `Object`, and we want to serialize that Object to the `CONTENT` column:



Example 15.14. Serialized Field Mapping

```
<class name="Article" table="ART">
  <field name="content" column="CONTENT" serialized="true"/>
  ...
</class>
```

15.11.3. Automatic Values

In [Section 15.5, “Datastore Identity” \[266\]](#), you saw how to use the `datastore-identity` element's `strategy` and `sequence` attributes to automatically generate datastore identity values. You can apply the same pattern to auto-generate values for fields of new objects. The `field` element's `value-strategy` attribute accepts the same strategies as `datastore-identity`'s `strategy` attribute. The `field` element also has a `sequence` attribute that functions exactly like `datastore-identity`'s `sequence` attribute. Let's modify our mappings to set `Article`'s `id` field automatically from the `ArticleSeq` sequence, and to make the `LineNumber.num` field auto-incrementing:

Example 15.15. Automatic Field Values

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <sequence name="ArticleSeq" datastore-sequence="ART_SEQ" />
    <class name="Article" table="ART">
      <!-- specifying the sequence attribute defaults the -->
      <!-- value-strategy attribute to "sequence" automatically -->
      <field name="id" column="ID" sequence="ArticleSeq"/>
      ...
    </class>
  </package>
  <package name="org.mag.subscribe">
    <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
      <field name="num" column="NUM" value-strategy="autoassign"/>
      ...
    </class>
  </package>
</orm>
```

Note

Kodo supports the standard `sequence`, `identity`, `autoassign`, `uuid-string`, and `uuid-hex` field value strategies. Kodo also offers the custom `version` strategy. Kodo will automatically set a field with the `version` value strategy to the object's optimistic version value. The field must be a numeric or date type. If you use a version field, you should not specify a separate version mapping (we covered version mapping in [Example 15.10, “Version Mapping” \[286\]](#)).

Kodo also defines a `read-only` metadata extension that allows you to ignore or restrict updates to a persistent field. See [Section 6.4.2.6, “Read-Only” \[486\]](#) in the Reference Guide for details. Fields that use the custom `version` value strategy are always read-only.

Using the `autoassign` or `identity` strategy on a field may cause the `PersistenceManager` to flush when retrieving the value of that field on a persistent-new instance. If the field is a primary key field, retrieving the object id of a persistent-new instance may also cause a flush, just as it does for new instances using the `autoassign` datastore identity strategy. See [Section 15.5, “Datastore Identity” \[266\]](#) for the complete explanation.

15.11.4. Secondary Tables

Sometimes a logical record is spread over multiple database tables. JDOR calls a class' declared table the *primary* table, and calls other tables that make up a logical record *secondary* tables. You can map any persistent field to a secondary table. Just write the standard field mapping, then perform these two additional steps:

1. Set the `field` element's `table` attribute to the name of the secondary table housing the mapped columns.
2. Nest a `join` element within the `field` that describes how the secondary table joins to the class' primary table. The `join` element goes before the field's nested `column` elements, if any. We covered joins in [Section 15.7, “Joins” \[269\]](#). In fact,

we already used the `join` element to join a subclass table to its superclass table in [Section 15.8.2, “new-table” \[275\]](#).

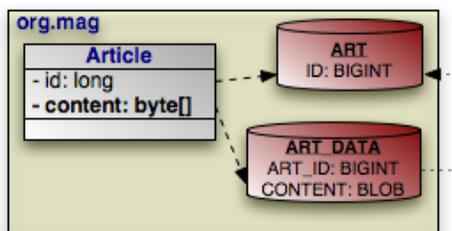
Secondary table joins, however, have one caveat that subclass table joins do not. In a subclass join, you know that there will always be a record in the superclass table for every record in the subclass table. In a secondary table join, that is not the case. Some databases are maintained such that primary table rows may not have matching secondary table rows. When you load objects based on these records, you want the fields mapped to the secondary table to come back `null`.

Joins that produce nulls for missing records are called *outer* joins. You may recall having read about another use for outer joins in [Section 15.9.3, “none” \[282\]](#). If your field is mapped to a secondary table that doesn't necessarily have a row for every primary table row, set the `join` element's `outer` attribute to `true`.

Note

When the `outer` attribute is `true`, Kodo will not insert null data rows into the named secondary table. That is, if you are inserting a new object, and it has `null` values for every field mapped to an outer-joined secondary table, Kodo will not insert a row into that secondary table.

In the following example, we move the `Article.content` field we mapped in [Section 15.11.2.2, “BLOB” \[291\]](#) into an outer-joined secondary table, like so:



Example 15.16. Secondary Table Field Mapping

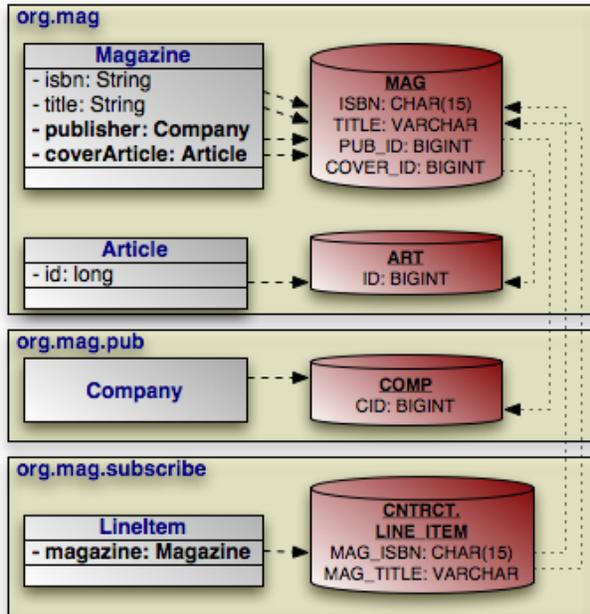
```
<class name="Article" table="ART">
  <field name="id" column="ID"/>
  <field name="content" table="ART_DATA" column="CONTENT">
    <join outer="true">
      <column name="ART_ID" target="ID"/>
    </join>
  </field>
  ...
</class>
```

Because the target table `ART` only has a single primary key column, we can use **join shortcuts** to get rid of the nested `column` element. Our final mapping becomes:

```
<class name="Article" table="ART">
  <field name="id" column="ID"/>
  <field name="content" table="ART_DATA" column="CONTENT">
    <join outer="true" column="ART_ID"/>
  </field>
  ...
</class>
```

15.11.5. Direct Relations

A direct relation is a non-embedded persistent field that holds a reference to another persistence-capable object. Our model has three direct relations: Magazine's publisher field is a direct relation to a Company, Magazine's coverArticle field is a direct relation to Article, and the LineItem.magazine field is a direct relation to a Magazine. Direct relations are represented in the database by foreign key columns:



You should be familiar with foreign key mapping from [Section 15.7, “Joins” \[269\]](#). Additionally, we have already demonstrated mapping foreign key columns onto the `join` element in [Section 15.8.2, “new-table” \[275\]](#) and [Section 15.11.4, “Secondary Tables” \[292\]](#). The only difference now is that direct relations map the foreign key columns directly onto the `field` element:

Example 15.17. Direct Relation Field Mapping

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      <field name="coverArticle">
        <column name="COVER_ID" target="ID"/>
      </field>
      <field name="publisher">
        <column name="PUB_ID" target="CID"/>
      </field>
      ...
    </class>
    <class name="Article" table="ART">
      <field name="id" column="ID"/>
      ...
    </class>
  </package>
  <package name="org.mag.pub">
    <class name="Company" table="COMP">
      <datastore-identity column="CID"/>
      ...
    </class>
  </package>
  <package name="org.mag.subscribe">
    <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
      <field name="magazine">
```

```

        <column name="MAG_ISBN" target="ISBN"/>
        <column name="MAG_TITLE" target="TITLE"/>
    </field>
    ...
</class>
...
</package>
</orm>

```

Or, after applying syntactic shortcuts:

```

<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      <field name="coverArticle" column="COVER_ID"/>
      <field name="publisher" column="PUB_ID"/>
      ...
    </class>
    <class name="Article" table="ART">
      <field name="id" column="ID"/>
      ...
    </class>
  </package>
  <package name="org.mag.pub">
    <class name="Company" table="COMP">
      <datastore-identity column="CID"/>
      ...
    </class>
  </package>
  <package name="org.mag.subscribe">
    <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
      <field name="magazine">
        <column name="MAG_ISBN" target="ISBN"/>
        <column name="MAG_TITLE" target="TITLE"/>
      </field>
      ...
    </class>
  </package>
</orm>

```

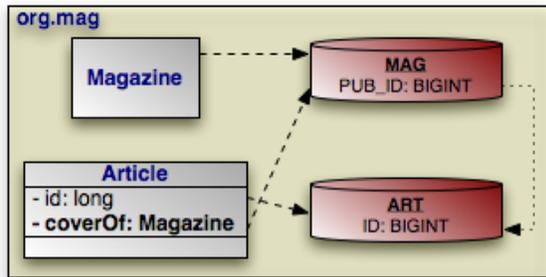
Note

A direct relation between types A and B is also called a *one-to-one* relation when every A has a unique B. It is called a *many-to-one* relation when multiple As can refer to the same B.

15.11.5.1. Inverse Keys

The direct relations above are all based on *forward* foreign keys. But relations can also be based on *inverse* keys. We described inverse foreign keys in [Section 15.7, “Joins” \[269\]](#).

For example, suppose that instead of `Magazine` having a relation to its cover `Article`, `Article` has a relation to the `Magazine` it serves as the cover *of*. We can represent this relation without changing our schema:



Example 15.18. Inverse Key Relation Field Mapping

Notice that the `Article.coverOf` field refers to the foreign key column in the `MAG` table by using the fully qualified column name.

```
<class name="Magazine" table="MAG">
  ...
</class>
<class name="Article" table="ART">
  <field name="id" column="ID"/>
  <field name="coverOf">
    <column name="MAG.PUB_ID" target="ID"/>
  </field>
  ...
</class>
```

After syntactic shortcuts:

```
<class name="Magazine" table="MAG">
  ...
</class>
<class name="Article" table="ART">
  <field name="id" column="ID"/>
  <field name="coverOf" column="MAG.PUB_ID"/>
  ...
</class>
```

15.11.5.2. Bidirectional Relations

In the previous sections, we saw that a single foreign key from the `MAG` table to the `ART` table can model an object relation from `Magazine` to `Article` (through the `Magazine.coverArticle` field), or from `Article` to `Magazine` (through the `Article.coverOf` field). A natural question is whether the foreign key can represent both relations at the same time. The answer is yes.

When two fields are logical inverses of each other like `Magazine.coverArticle` and `Article.coverOf`, they form a *bidirectional relation*. And when the two fields of a bidirectional relation share the same database mapping, JDOR formalizes the connection with the `mapped-by` field attribute. Using the `mapped-by` attribute, we can map both `Magazine.coverArticle` and `Article.coverOf` as follows:

Example 15.19. Mapping a Bidirectional Relation

```

<class name="Magazine" table="MAG">
  <field name="coverArticle" column="PUB_ID"/>
  ...
</class>
<class name="Article" table="ART">
  <field name="coverOf" mapped-by="coverArticle"/>
  ...
</class>

```

Marking `Article.coverOf` as `mapped-by Magazine.coverArticle` means two things:

1. `Article.coverOf` uses the foreign key mapped by `Magazine.coverArticle`, but inverts it. In fact, it is illegal to specify any additional mapping information when you use the `mapped-by` attribute. All mapping information is read from the referenced field.
2. `Magazine.coverArticle` is the "owner" of the relation. The field that specifies the mapping data is always the owner. This means that changes to the `Magazine.coverArticle` field are reflected in the database, while changes to the `Article.coverOf` field alone are not. Changes to `Article.coverOf` may still affect the JDOR implementation's cache, however. Thus, it is very important that you keep your object model consistent by properly maintaining both sides of your bidirectional relations at all times.

Note

It is more efficient to make the field that maps to the foreign key's natural *forward* direction the owner of a bidirectional relation. Specify the mapping data on the forward foreign key side, and use `mapped-by` on the inverse foreign key side, just as we did with `Magazine.coverArticle` and `Article.coverOf`.

You should always take advantage of the `mapped-by` attribute rather than mapping each field of a bidirectional relation independently. Failing to do so may result in the JDOR implementation trying to update the database with conflicting data. Be careful to only mark one side of the relation as `mapped-by`, however. One side has to actually do the mapping!

Note

You can configure Kodo to automatically synchronize both sides of a bidirectional relation, or to perform various actions when it detects inconsistent relations. See [Section 5.4, "Managed Inverses" \[458\]](#) in the Reference Guide for details.

15.11.6. Basic Collections

Collections and arrays of primitive wrappers, `Strings`, `Dates`, or anything else that can be directly stored in a database column all fall under the umbrella of basic collection mapping. Basic collection mapping is just a special case of secondary table mapping, as seen in [Section 15.11.4, "Secondary Tables" \[292\]](#). Basic collections map to a secondary table consisting of three components:

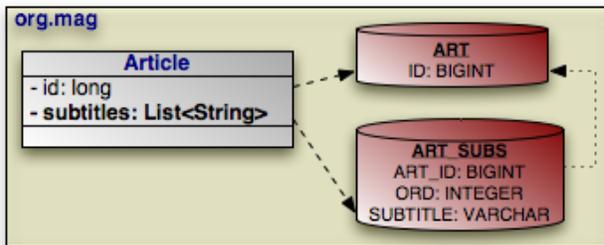
1. Foreign key columns linking back to the owning class' primary table. As with all secondary table mappings, these foreign key columns are mapped to a `join` element.
2. A column to hold a collection element. Each row of the collection table holds a single collection or array element. Logically enough, the `element` element represents a collection or array element.
3. An optional ordering column. Relational databases do not preserve record order. This column stores the relative position of

elements within the collection or array so that the JDOR implementation can retrieve them in the same order they were in when stored. The order element anchors ordering column information in mapping metadata.

Note

In addition to supporting an ordering column, Kodo allows you to order on the collection element values, or, in the case of relations, fields of the related type. See [Section 6.4.2.3, “Order-By” \[485\]](#) in the Reference Guide for details.

The `Article.subtitles` field is the only basic collection in our model. This List of Strings is mapped as follows:



In mapping metadata, this becomes:

Example 15.20. Basic Collection Field Mapping

```
<class name="Article" table="ART">
  <field name="id" column="ID"/>
  <field name="subtitles" table="ART_SUBS">
    <join>
      <column name="ART_ID" target="ID"/>
    </join>
    <element column="SUBTITLE"/>
    <order column="ORD"/>
  </field>
  ...
</class>
```

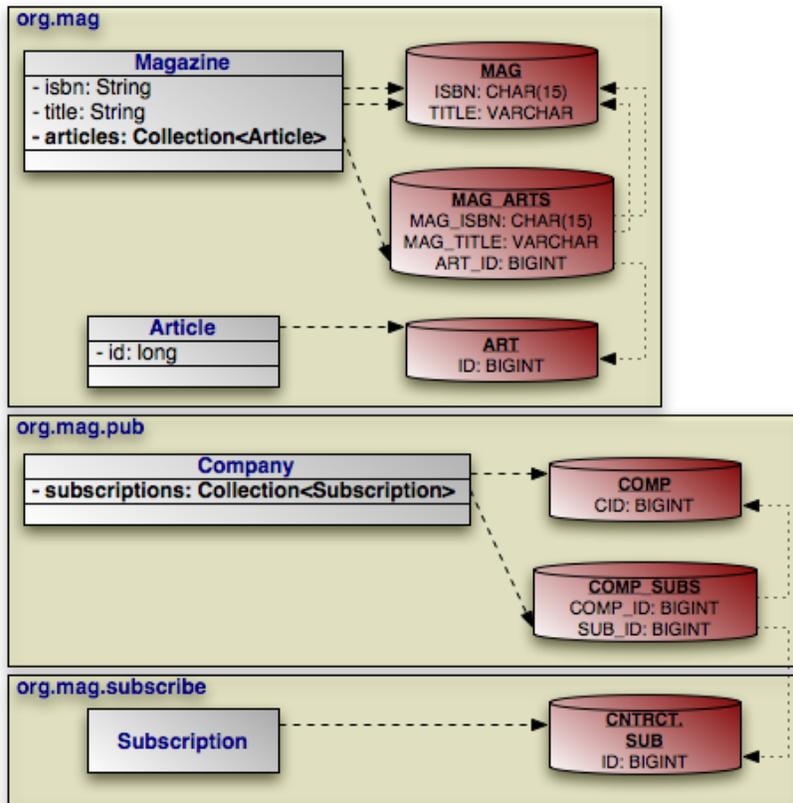
Taking advantage of **join shortcuts** yields:

```
<class name="Article" table="ART">
  <field name="id" column="ID"/>
  <field name="subtitles" table="ART_SUBS">
    <join column="ART_ID"/>
    <element column="SUBTITLE"/>
    <order column="ORD"/>
  </field>
  ...
</class>
```

The same pattern applies to mapping collections of BLOBs or CLOBs, or any elements that can be stored in one or more data columns.

15.11.7. Association Table Collections

An *association table* consists of two foreign keys, plus an optional ordering column. In other words, if you replace the data columns of a basic collection table (Section 15.11.6, “Basic Collections” [297]) with another foreign key, you produce an association table. As its name implies, each row of an association table associates two objects together. JDOR uses association tables to represent collections of persistence-capable objects: one foreign key refers back to the collection's owner, and the other refers to a collection element. Our model's `Magazine.articles` and `Company.subscriptions` fields both have association table mappings.



Note

An association table relation between types A and B is also called a *one-to-many* relation when every A has a unique set of Bs. It is called a *many-to-many* relation when multiple As can refer to the same B.

In metadata, an association table mapping is exactly like a basic collection mapping (Section 15.11.6, “Basic Collections” [297]), except that the `element` columns represent a foreign key. You should be very familiar with foreign key mapping by now. See Section 15.7, “Joins” [269] for a refresher.

Example 15.21. Association Table Collection Field Mapping

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      <field name="articles" table="MAG_ARGS">
        <join>
          <column name="MAG_ISBN" target="ISBN"/>
        </join>
      </field>
    </class>
  </package>

```

```

        <column name="MAG_TITLE" target="TITLE"/>
    </join>
    <element>
        <column name="ART_ID" target="ID"/>
    </element>
</field>
...
</class>
<class name="Article" table="ART">
    <field name="id" column="ID"/>
    ...
</class>
...
</package>
<package name="org.mag.pub">
    <class name="Company" table="COMP">
        <datastore-identity column="CID" strategy="autoassign"/>
        <field name="subscriptions" table="COMP_SUBS">
            <join>
                <column name="COMP_ID" target="CID"/>
            </join>
            <element>
                <column name="SUB_ID" target="ID"/>
            </element>
        </field>
    </class>
    ...
</package>
<package name="org.mag.subscribe">
    <class name="Subscription" table="CNTRCT.SUB">
        <field name="Contract.id" column="ID"/>
    </class>
    ...
</package>
</orm>

```

Applying **join shortcuts** yields:

```

<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      <field name="articles" table="MAG_ARGS">
        <join>
          <column name="MAG_ISBN" target="ISBN"/>
          <column name="MAG_TITLE" target="TITLE"/>
        </join>
        <element column="ART_ID"/>
      </field>
    </class>
    <class name="Article" table="ART">
      <field name="id" column="ID"/>
    </class>
  </package>
  <package name="org.mag.pub">
    <class name="Company" table="COMP">
      <datastore-identity column="CID" strategy="autoassign"/>
      <field name="subscriptions" table="COMP_SUBS">
        <join column="COMP_ID"/>
        <element column="SUB_ID"/>
      </field>
    </class>
  </package>
  <package name="org.mag.subscribe">
    <class name="Subscription" table="CNTRCT.SUB">
      <field name="Contract.id" column="ID"/>
    </class>
  </package>
</orm>

```

None of our model's association table collections are ordered, but if they were, we could use the `order` mapping element to specify the ordering column, just as in a **basic collection** mapping.

15.11.7.1. Bidirectional Relations

Association tables have no built-in sense of direction. Each row links two objects, period. You can look at the `MAG_ARTS` table in our example above as linking each `Magazine` to its `Articles`, or as linking each `Article` to the `Magazines` it appears in. The two outlooks are equally valid. This makes association tables perfect candidates for representing *bidirectional relations*.

Section 15.11.5.2, “Bidirectional Relations” [296] examined direct bidirectional relations that share a foreign key. Association table bidirectional relations share association table rows instead. Aside from that, all the same concepts apply. In particular, you still use the `mapped-by` attribute to mark one collection field as mapped and "owned" by another collection field. The owning field is responsible for manipulating association table records - the same caveats about keeping your bidirectional relations synchronized apply equally to collections.

In the example below, we have added an `Article.magazines` field as the logical inverse of the the `Magazine.articles` field.

Example 15.22. Mapping Bidirectional Association Table Relations

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      <field name="articles" table="MAG_ARGS">
        <join>
          <column name="MAG_ISBN" target="ISBN"/>
          <column name="MAG_TITLE" target="TITLE"/>
        </join>
        <element column="ART_ID"/>
      </field>
      ...
    </class>
    <class name="Article" table="ART">
      <field name="id" column="ID"/>
      <field name="magazines" mapped-by="articles"/>
      ...
    </class>
    ...
  </package>
</orm>
```

Note

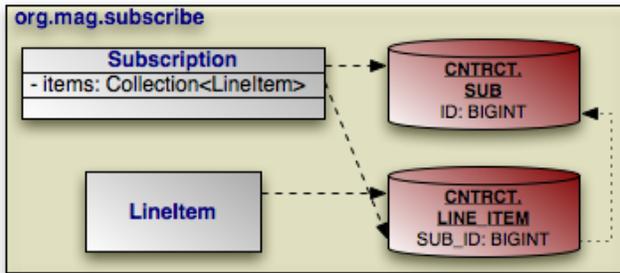
Association tables are symmetric; therefore, the choice of owning vs. non-owning side is arbitrary. However, only the owning side of the relation can order the its collection elements.

15.11.8. Inverse Key Collections

Inverse key collection mappings are the only standard collection mappings that do not use a secondary table. Instead, the mapping is based on an inverse foreign key in the table of the related class. Rather than try to explain the mapping abstractly, let's jump to an example.

Note

Inverse key collection relations are also called *one-to-many* relations.



The `Subscription.items` field in our model is an inverse key collection. `Subscription.items` is a `Collection` of `LineItem` objects. There is a foreign key from `LineItem`'s table, `CNTRCT.LINE_ITEM`, to `Subscription`'s table, `CNTRCT.SUB`. The relation is loaded by selecting all `LineItem` records whose foreign key links back to the owning `Subscription`'s record.

This situation may look familiar to you. It is, in fact, exactly like the inverse key direct relation from **Section 15.11.5.1, “Inverse Keys” [295]**. The only thing separating this mapping from the inverse key direct relation mapping is that in this case, multiple rows in `CNTRCT.LINE_ITEM` might have a foreign key to the same `CNTRCT.SUB` record. At the object level, this results in a collection rather than a direct reference.

This difference is also reflected in the mapping metadata. While direct references are mapped to `field`, collection elements are mapped to `field`'s nested `element` element. Also, you do not have to fully qualify the name of the foreign key columns in an inverse key collection, as you do in an inverse key direct relation. That is because the lack of a secondary table is a hint to the JDOR implementation that you are using an inverse key collection, and it knows to look in the table of the related class for the foreign key columns.

Example 15.23. Inverse Key Collection Field Mapping

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag.subscribe">
    <class name="Subscription" table="CNTRCT.SUB">
      <field name="Contract.id" column="ID"/>
      <field name="items">
        <element>
          <column name="SUB_ID" target="ID"/>
        </element>
      </field>
      ...
    </class>
    <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
      <field name="Contract.id" column="ID"/>
      ...
    </class>
    ...
  </package>
</orm>
```

Or with **join shortcuts** in place:

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag.subscribe">
    <class name="Subscription" table="CNTRCT.SUB">
      <field name="Contract.id" column="ID"/>
      <field name="items">
        <element column="SUB_ID"/>
      </field>
    </class>
  </package>
</orm>
```

```

    ...
  </class>
  <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
    <field name="Contract.id" column="ID"/>
    ...
  </class>
  ...
</package>
</orm>

```

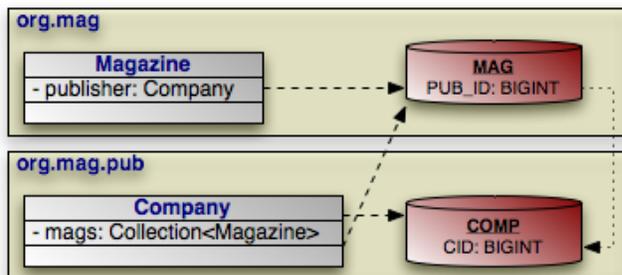
Though it is rare, you can map an order column to an inverse key collection to maintain collection order. You use the `order` mapping element, as in any other collection mapping. The order column must be in the same table as the foreign key column(s).

Note

Kodo allows you to specify an order column even in bidirectional inverse key collection mappings.

15.11.8.1. Bidirectional Relations

Most inverse key collections are actually half of a bidirectional relation. The other half is a standard direct relation mapping ([Section 15.11.5, “Direct Relations” \[294\]](#)) onto the foreign key. In fact, it is unusual to write an inverse key collection mapping as we did above; the vast majority of inverse key collections simply use the `mapped-by` attribute introduced in [Section 15.11.5.2, “Bidirectional Relations” \[296\]](#) to refer to their partner field. Our model's `Magazine.publisher` and `Company.magazines` fields are perfect illustrations of the normal pairing between a direct relation and an inverse key collection.



Example 15.24. Direct Relation / Inverse Key Collection Pair

```

<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="publisher" column="PUB_ID"/>
      ...
    </class>
  </package>
  <package name="org.mag.pub">
    <class name="Company" table="COMP">
      <datastore-identity column="CID"/>
      <field name="mags" mapped-by="publisher"/>
      ...
    </class>
  </package>
</orm>

```

15.11.9. Maps

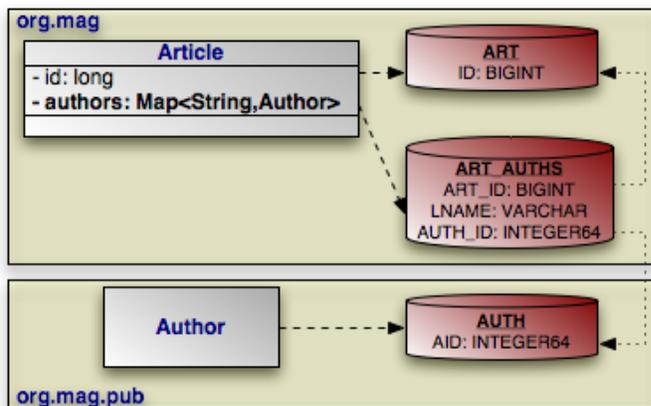
JDOR supports maps of all kinds. Your map keys can be basic types like `Strings`, more interesting types like `BLOBs` and `CLOBs`, or even persistence-capable objects. Map values have equal flexibility. Rather than treat each combination of key and value types as a separate mapping, this section explains the concepts of map mapping in JDOR. Armed with a solid understanding of map concepts, you will be able to write a mapping for any combination of key and value classes, because object-relational mapping in JDOR is remarkably consistent.

There are two ways to represent a map. The first is to write the map to a secondary table. You may recall secondary table mapping of singular values from [Section 15.11.4, “Secondary Tables” \[292\]](#). In a map's secondary table, each row represents a map entry. Therefore, every map's secondary table has the following three components:

1. Foreign key columns linking back to the owning class' primary table. As with all secondary table mappings, these foreign key columns are mapped to a `join` element.
2. Columns to hold the entry's key. If the map uses a simple key type like `String`, these columns will be standard data columns. If the map uses persistence-capable objects as keys, these will be foreign key columns linking to the records for those objects. Key columns are anchored to a `key` element nested in the `field` element.
3. Columns to hold the entry's value. Like key columns, value columns can be direct data columns or foreign key columns, depending on the map's value type. You specify map value columns on the `field`'s `value` element.

In short, secondary table map mapping is like a combination of basic collection mapping ([Section 15.11.6, “Basic Collections” \[297\]](#)) and association table collection mapping ([Section 15.11.7, “Association Table Collections” \[298\]](#)), depending on the key and value types involved.

`Article.authors` is the lone map field in our model. For each author of the article, this field maps the author's last name to the persistent `Author` instance.



To translate this to mapping metadata, we perform a basic secondary table mapping, then add the key columns to the `key` element and the value foreign key columns to the `value` element.

Example 15.25. Map Field Mapping

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Article" table="ART">
      <field name="id" column="ID"/>
      <field name="authors" table="ART_AUTHS">
        <join>
          <column name="ART_ID" target="ID"/>

```

```

        </join>
        <key column="LNAME"/>
        <value>
            <column name="AUTH_ID" target="AID"/>
        </value>
    </field>
    ...
</class>
...
</package>
<package name="org.mag.pub">
    <class name="Author" table="AUTH">
        <datastore-identity sequence="AuthorSeq">
            <column name="AID" sql-type="INTEGER64"/>
        </datastore-identity>
    </class>
    ...
</package>
</orm>

```

Or with **join shortcuts** in place:

```

<?xml version="1.0"?>
<orm>
    <package name="org.mag">
        <class name="Article" table="ART">
            <field name="id" column="ID"/>
            <field name="authors" table="ART_AUTHS">
                <join column="ART_ID"/>
                <key column="LNAME"/>
                <value column="AUTH_ID"/>
            </field>
            ...
        </class>
        ...
    </package>
    <package name="org.mag.pub">
        <class name="Author" table="AUTH">
            <datastore-identity>
                <column name="AID" sql-type="INTEGER64"/>
            </datastore-identity>
            ...
        </class>
        ...
    </package>
</orm>

```

The second way to represent a map in JDOR is as a set of persistence-capable values with derived keys. In the example above, the key for each `Author` is her last name. So rather than create a secondary table with a dedicated key column, we can simply map the `Author` values, then declare that the key is mapped by each `Author`'s last name. This lets us get rid of the key column altogether.

Example 15.26. Derived Key Mapping

```

<?xml version="1.0"?>
<orm>
    <package name="org.mag">
        <class name="Article" table="ART">
            <field name="id" column="ID"/>
            <field name="authors" table="ART_AUTHS">
                <join column="ART_ID"/>
                <key mapped-by="lastName"/>
                <value column="AUTH_ID"/>
            </field>
            ...
        </class>
        ...
    </package>
    <package name="org.mag.pub">
        <class name="Author" table="AUTH">

```

```

        <datastore-identity>
          <column name="AID" sql-type="INTEGER64"/>
        </datastore-identity>
        ...
      </class>
      ...
    </package>
  </orm>

```

When the key is derived from a field of the value type, you aren't limited to using a secondary table to hold the map. In fact, you can take any mapping that works for a collection of persistence-capable objects and turn it into a map mapping in two steps:

1. Add a key XML element. Set its `mapped-by` attribute to the name of a field in the persistence-capable value class. JDOR will automatically map each entry on the value of this field.
2. Use the `value` XML element instead of the `element` collection element to map the related persistence-capable objects.

The diagram above depicts the mapping of a `Subscription`'s `LineItems`. We created this mapping in [Section 15.11.8, “Inverse Key Collections” \[301\]](#). The example below turns this mapping into a map. Each map entry associates a `LineItem`'s `num` field value to that `LineItem`.

Example 15.27. Inverse Foreign Key Map Mapping

```

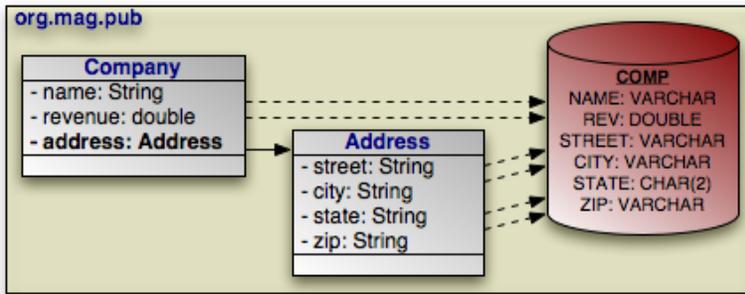
public class Subscription
  extends Contract
{
  private Map<Long,LineItem> lineItems;
  ...
}

<?xml version="1.0"?>
<orm>
  <package name="org.mag.subscribe">
    <class name="Subscription" table="CNTRCT.SUB">
      <field name="Contract.id" column="ID"/>
      <field name="items">
        <key mapped-by="num"/>
        <value column="SUB_ID"/>
      </field>
      ...
    </class>
    <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
      <field name="Contract.id" column="ID"/>
      ...
    </class>
    ...
  </package>
</orm>

```

15.11.10. Embedded Objects

[Chapter 5, Metadata \[193\]](#) describes JDO's concept of *embedded* objects. The field values of embedded objects are stored as part of the owning record, rather than as a separate database record. Thus, instead of mapping a relation to an embedded object as a foreign key, you map all the fields of the embedded instance to columns in the owning field's table.



To perform an embedded mapping, nest the embedded element in the embedded field. This element has the following attributes:

- null-indicator-column:** Set this attribute to the name of a column that indicates whether the embedded instance is null. When the JDOR implementation retrieves the embedded object, it will check this column first. If the column value is null, the JDOR runtime will load null into the field that owns the embedded object. If the column value is not null, the JDOR runtime will set the owning field to a new embedded object instance, and will load that instance's fields with data from their mapped columns.

The null indicator column can be a column that is also mapped to an embedded field, or can be an artificial column whose sole purpose is to indicate whether the embedded instance is null. In the latter case, the JDOR implementation will set this column automatically.

If you do not specify a null indicator column, then an embedded instance will never get loaded as null. Even if you commit an owning object with its embedded field set to null, the next time you re-read the owning object its embedded field will get loaded with a non-null embedded instance. The embedded instance will have default values for all of its fields.

Within the embedded element, nest field elements for all the fields of the embedded object. Map these fields as you would any other fields; you can even map recursive embedded fields, to arbitrary depth.

In our model, Company embeds its Address, as pictured in the diagram above. Author also embeds its Address. Let's see how this looks in mapping metadata:

Example 15.28. Embedded Field Mapping

This example uses the same column names for Address's embedded fields in both the COMP and AUTH tables. Note, though, that nothing prevents you from using completely different mappings each time you embed an object.

```

<?xml version="1.0"?>
<orm>
  <package name="org.mag.pub">
    <class name="Company" table="COMP">
      <field name="address">
        <embedded null-indicator-column="STREET">
          <field name="street" column="STREET"/>
          <field name="city" column="CITY"/>
          <field name="state">
            <column name="STATE" jdbc-type="char" length="2"/>
          </field>
          <field name="zip" column="ZIP"/>
        </embedded>
      </field>
      ...
    </class>
    <class name="Author" table="AUTH">
      <field name="address">
        <embedded null-indicator-column="STREET">
          <field name="street" column="STREET"/>
          <field name="city" column="CITY"/>
          <field name="state">
            <column name="STATE" jdbc-type="char" length="2"/>
          </field>
        </embedded>
      </field>
    </class>
  </package>
</orm>

```

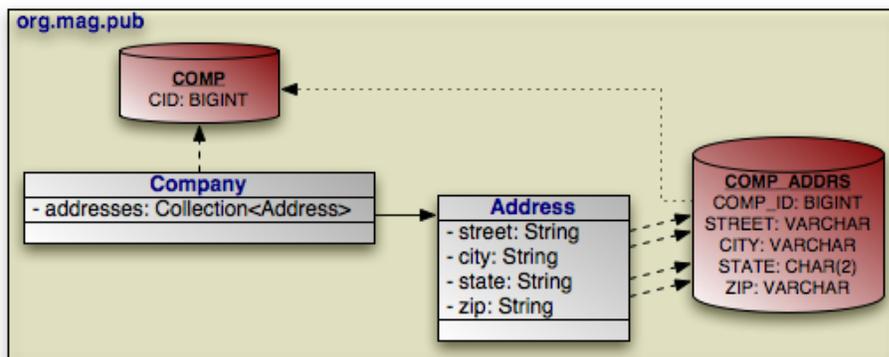
```

        </field>
        <field name="zip" column="ZIP"/>
      </embedded>
    </field>
    ...
  </class>
  ...
</package>
</orm>

```

We set the null indicator column to STREET. Any time the STREET column is null, JDOR loads null into the address field. Any time STREET is not null, JDOR loads a new Address instance into the address field, and populates that Address with data based on the above field mappings.

This section focused on fields that hold a direct reference to an embedded object. However, JDOR also supports collections that contain embedded object elements, and maps that contains embedded object keys, values, or both. In these cases, the columns for the embedded object's fields are part of the collection or map table. Mapping embedded elements, keys, or values is exactly like mapping an embedded field, except that instead of nesting the embedded element under the field element, you nest it under the element, key, or value elements, respectively. For example, if Company had a collection of embedded addresses, like so:



Then the mapping would be:

Example 15.29. Embedded Collection Elements

```

<class name="Company" table="COMP">
  <datastore-identity column="CID" strategy="autoassign"/>
  <field name="addresses" table="COMP_ADDRs">
    <join column="COMP_ID"/>
    <element>
      <embedded>
        <field name="street" column="STREET"/>
        <field name="city" column="CITY"/>
        <field name="state">
          <column name="STATE" jdbc-type="char" length="2"/>
        </field>
        <field name="zip" column="ZIP"/>
      </embedded>
    </element>
  </field>
  ...
</class>

```

Normally, you will not use a null indicator column for embedded collection elements, map keys, and map values. Only do so if you plan on storing null values in your collection or map.

15.12. Foreign Keys

Section 15.7, “Joins” [269] introduced *foreign keys* as logical links between records. Most relational databases have the ability to manifest these logical links as physical database constraints. Foreign key constraints have a *delete action* that runs if the linked-to record is deleted, and an *update action* that runs if the linked-to record's primary key changes. You have a choice between the following action types:

- `restrict`: Throw an exception if the linked-to record is deleted/updated. All databases with physical foreign keys support this action. Using this action ensures that there are no orphaned references in the database, because a record cannot be deleted or change its primary key while there are still foreign keys referencing it.
- `cascade`: Modify the foreign key record to match the linked-to record. As a delete action, this means that if the referenced row is deleted, the row referencing it will be deleted as well. As an update action, it means that the referencing row's foreign key values will automatically change whenever the linked-to row's primary key values are modified. Not all databases support cascading foreign keys, especially cascading update action keys.
- `null`: If the referenced row is deleted or changes its primary key, null the columns of this foreign key. Some databases do not support this action.
- `default`: Set the columns of this foreign key to their default values if the linked-to row is deleted or changes its primary key. Some databases do not support this action.

JDOR uses foreign key declarations both at table-creation time and at runtime. If your JDOR vendor provides a tool for creating tables based on your mappings, then that tool will create physical database foreign keys corresponding to your foreign key declarations. At runtime, the JDOR implementation can use your foreign key markup to order its `INSERT`, `UPDATE`, and `DELETE` statements so as not to violate any foreign key constraints. For example, if you are deleting two records A and B, where A has a `restrict-delete` action foreign key to B, your JDOR implementation must issue the `DELETE` for A before the `DELETE` for B. Otherwise, A's foreign key will throw an exception when B is deleted, aborting the transaction. Thus, it is important that you faithfully represent your tables' foreign keys in your mapping metadata.

Note

If your mapping metadata specifies foreign key actions that are not supported by your database, Kodo's table creation tools fall back to supported actions automatically. If your database does not support foreign keys at all, Kodo's tools will not generate any foreign key creation SQL.

JDOR provides multiple ways to represent a physical foreign key. The easiest way is to add the `delete-action` attribute to any mapping element holding foreign key columns. This attribute accepts any of the foreign key action names listed above, and represents a foreign key with the `restrict` update action and the given delete action. Because the vast majority of foreign keys use the `restrict` update action, setting a delete action is an easy and concise way to represent most foreign keys. If your physical foreign key is on the columns of a relation field, set the `delete-action` attribute on the `field` element. If your foreign key is on the join columns between a subclass table and superclass table, add the `delete-action` attribute to the `inheritance` element's nested `join` element. The following elements can all house join columns, and therefore all accept the `delete-action` attribute: `field`, `join`, `element`, `key`, `value`.

Note

If your vendor's schema creation tools generate physical foreign keys on relations by default, setting the `delete-action` attribute to `none` will suppress foreign key creation.

Example 15.30. Using the delete-action Attribute

The link from a `Magazine` to its cover `Article`, the join from `MAG_ARTS` association table records to their owning `Magazines`, and the reference from each association table record to its `Article` are all given physical foreign keys below. With these foreign keys in place, attempting to delete any `Article` still referenced by a `Magazine` will throw an exception on flush. Deleting a `Magazine` will automatically clear its entries in the `MAG_ARTS` table (the JDOR implementation should do this anyway, but having a cascading foreign key in place is better in case other, less robust processes delete records without cleaning up after themselves).

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      <field name="coverArticle" column="COVER_ID" delete-action="restrict"/>
      <field name="articles" table="MAG_ARTS">
        <join delete-action="cascade">
          <column name="MAG_ISBN" target="ISBN"/>
          <column name="MAG_TITLE" target="TITLE"/>
        </join>
        <element column="ART_ID" delete-action="restrict"/>
      </field>
      ...
    </class>
    <class name="Article" table="ART">
      <field name="id" column="ID"/>
      ...
    </class>
    ...
  </package>
</orm>
```

The second way to represent a physical foreign key is with a contextual `foreign-key` element. You can nest the `foreign-key` element within any element that accepts a `delete-action` attribute. The `foreign-key` element goes just after any nested `column` elements. And just as nested `column` elements provide more power than the `column` attribute, nested `foreign-key` elements provide more power than the `delete-action` attribute. The `foreign-key` element has the following attributes:

- `name`: The name of the foreign key. This attribute is only used during table creation. If it is not present, the JDOR implementation will choose a suitable default name, or create an unnamed key.
- `delete-action`: The key's delete action. Defaults to `restrict`.
- `update-action`: The key's update action. Defaults to `restrict`.
- `deferred`: Set this attribute to `true` if the actions of this key are *deferred*. Deferred actions wait until just before the database transaction commits before taking place. This simplifies things for the JDOR implementation, because it no longer has to carefully order its inserts, updates, and deletes to avoid foreign key constraints. Unfortunately, many databases do not support deferred constraints.

The `foreign-key` element also allows nested `extension` elements.

Here is the previous example, modified to use `foreign-key` elements rather than `delete-action` attributes. This version also specifies the names of the keys, and makes some of them deferred.

Example 15.31. Using Contextual Foreign Key Elements

```

<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      <field name="coverArticle" column="COVER_ID">
        <foreign-key name="COV_ART_FK" deferred="true"/>
      </field>
      <field name="articles" table="MAG_ARTS">
        <join column="MAG_ID">
          <foreign-key name="OWNER_MAG_FK" delete-action="cascade"/>
        </join>
        <element column="ART_ID">
          <foreign-key name="MAG_ART_FK" deferred="true"/>
        </element>
      </field>
      ...
    </class>
    <class name="Article" table="ART">
      <field name="id" column="ID"/>
      ...
    </class>
    ...
  </package>
</orm>

```

15.13. Indexes

Indexes are database structures that speed up searches, similar to how the index of a book helps you find what you're looking for. Placing indexes on columns that you often use in queries can drastically improve query performance. Note that primary key columns and columns with a physical foreign key are already naturally indexed by the database; you do not need to place additional indexes on them.

JDOR only uses index markup for table creation; it is not used at runtime. If you are mapping your classes to existing tables, you do not have to denote your existing indexes in the mapping metadata.

As with **foreign keys**, JDOR allows you to specify an index with either an attribute or an element. In the attribute-based approach, you place the `indexed` attribute on the mapping element whose columns you want to index. This attribute recognizes the values `true`, `false`, and `unique`. A *unique* index assumes that the combined data in the index's columns is unique for each row. The following mapping elements accept the `indexed` attribute: `discriminator`, `version`, `field`, `join`, `element`, `key`, `value`, `order`.

Example 15.32. Using the `indexed` Attribute

Indexing the columns of the `Article.title` field:

```

<class name="Article" table="ART">
  <field name="title" column="TITLE" indexed="true"/>
  ...
</class>

```

In place of the `indexed` attribute, you can use a contextual `index` element. Any element that accepts the `indexed` attribute also accepts a nested `index` element. The element goes after any nested `columns` and `foreign-keys`. It has the following attributes:

- `name`: The name of the index. If you do not specify a name, the JDOR implementation will choose a suitable default.
- `unique`: Boolean attribute indicating whether this is a unique index.

The `index` element also permits nested extension elements.

Here is the previous example with an `index` element in place of the indexed attribute:

Example 15.33. Using Contextual Index Elements

```
<class name="Article" table="ART">
  <field name="title" column="TITLE">
    <index name="ART_TTL_IDX"/>
  </field>
  ...
</class>
```

15.14. Unique Constraints

Unique constraints ensure that the data in a column or combination of columns is unique for each row. A table's primary key, for example, functions as an implicit unique constraint. In JDOR, you represent other unique constraints with the `unique` attribute or the `unique` element. These attributes and elements are used during table creation to generate the proper database constraints, and may also be used at runtime to order `INSERT`, `UPDATE`, and `DELETE` statements. For example, suppose there is a unique constraint on the columns of field `F`. In the same JDO transaction, you delete an object `A` and make persistent a new object `B`, both with the same `F` value. The JDOR runtime must ensure that the SQL deleting `A` is sent to the database before the SQL inserting `B` to avoid a unique constraint violation.

The `unique` attribute can be either `true` or `false`. When placed on a mapping element, it applies to all columns of that element. The following elements accept the `unique` attribute: `field`, `join`, `element`, `key`, `value`.

Example 15.34. Using the unique Attribute

The following adds a unique constraint to the columns of the `Article.title` field:

```
<class name="Article" table="ART">
  <field name="title" column="TITLE" unique="true"/>
  ...
</class>
```

The contextual `unique` element serves the same purpose as the `unique` attribute, but allows you to specify more information about the constraint. Any element that accepts the `unique` attribute also accepts a nested `unique` element. The element goes after any nested `columns`, `foreign-keys`, and `indexes`. The `unique` element has the following attributes:

- `name`: The constraint name. If left unspecified, the JDOR implementation will choose a suitable default or create an unnamed constraint.
- `deferred`: Set this attribute to `true` if this constraint's actions are *deferred*. Deferred actions wait until just before

the database transaction commits before running. This means the JDOR implementation doesn't have to worry about ordering its SQL statements to meet constraints. Unfortunately, many databases do not support deferred constraints.

The unique element also allows nested extension elements.

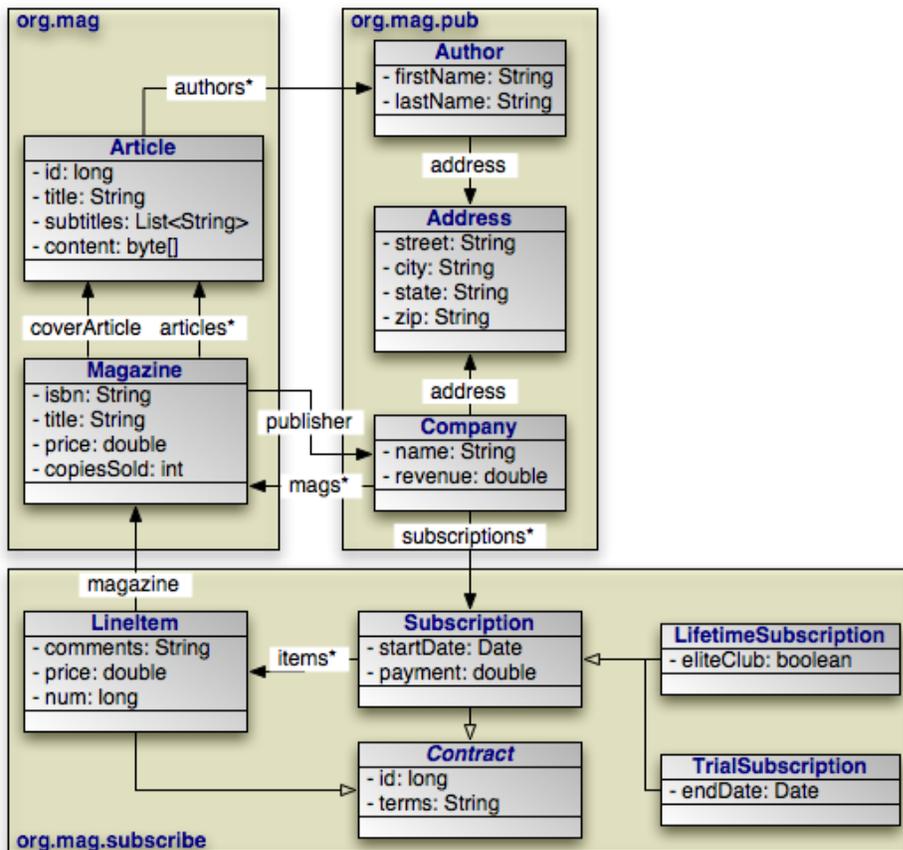
Here is our previous example using a unique element instead of an attribute:

Example 15.35. Using Contextual Unique Elements

```
<class name="Article" table="ART">
  <field name="title" column="TITLE">
    <unique name="ART_TTL_UNQ"/>
  </field>
  ...
</class>
```

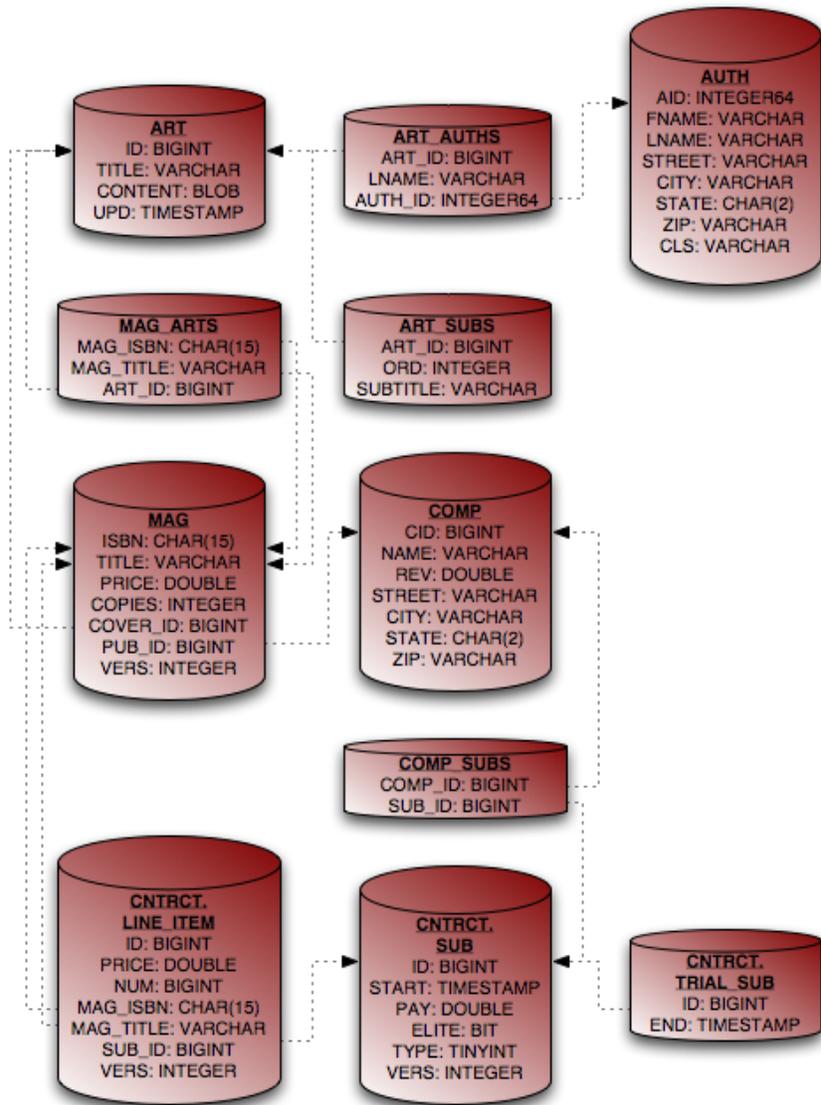
15.15. The Complete Document

We began this chapter with the goal of mapping the following object model:



That goal has now been met. In the course of explaining JDOR's object-relational mapping metadata, we slowly built the requisite

schema and mappings for the complete model. First, the database schema:



And finally, the complete JDOR mapping document:

Example 15.36. Full Mapping Document

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <sequence name="ArticleSeq" datastore-sequence="ART_SEQ"/>
    <class name="Magazine" table="MAG">
      <version column="VERS" strategy="version-number"/>
      <field name="isbn">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE"/>
      <field name="price" column="PRICE"/>
      <field name="copiesSold" column="COPIES"/>
      <field name="coverArticle" column="COVER_ID" delete-action="restrict"/>
      <field name="articles" table="MAG_ARGS">
        <join delete-action="cascade">
          <column name="MAG_ISBN" target="ISBN"/>
        </join>
      </field>
    </class>
  </package>

```

```

        <column name="MAG_TITLE" target="TITLE"/>
    </join>
    <element column="ART_ID" delete-action="restrict"/>
</field>
<field name="publisher" column="PUB_ID" delete-action="restrict"/>
</class>
<class name="Article" table="ART">
    <version column="UPD" strategy="date-time"/>
    <field name="id" column="ID" sequence="ArticleSeq"/>
    <field name="title" column="TITLE" unique="true"/>
    <field name="content" column="CONTENT"/>
    <field name="authors" table="ART_AUTHS">
        <join column="ART_ID" delete-action="cascade"/>
        <key column="LNAME"/>
        <value column="AUTH_ID" delete-action="restrict"/>
    </field>
    <field name="subtitles" table="ART_SUBS">
        <join column="ART_ID" delete-action="cascade"/>
        <element column="SUBTITLE"/>
        <order column="ORD"/>
    </field>
</class>
</package>
<package name="org.mag.pub">
    <sequence name="AuthorSeq" factory-class="Author$SequenceFactory"/>
    <class name="Company" table="COMP">
        <datastore-identity column="CID" strategy="autoassign"/>
        <version strategy="none"/>
        <field name="name" column="NAME" unique="true"/>
        <field name="revenue" column="REV"/>
        <field name="address">
            <embedded null-indicator-column="STREET">
                <field name="street" column="STREET"/>
                <field name="city" column="CITY"/>
                <field name="state">
                    <column name="STATE" jdbc-type="char" length="2"/>
                </field>
                <field name="zip" column="ZIP" indexed="true"/>
            </embedded>
        </field>
        <field name="subscriptions" table="COMP_SUBS">
            <join column="COMP_ID" delete-action="cascade"/>
            <element column="SUB_ID" delete-action="restrict"/>
        </field>
        <field name="mags" mapped-by="publisher"/>
    </class>
    <class name="Author" table="AUTH">
        <datastore-identity sequence="AuthorSeq">
            <column name="AID" sql-type="INTEGER64"/>
        </datastore-identity>
        <inheritance>
            <discriminator column="CLS" strategy="class-name"/>
        </inheritance>
        <version strategy="state-comparison"/>
        <field name="firstName" column="FNAME"/>
        <field name="lastName" column="LNAME" indexed="true"/>
        <field name="address">
            <embedded null-indicator-column="STREET">
                <field name="street" column="STREET"/>
                <field name="city" column="CITY"/>
                <field name="state">
                    <column name="STATE" jdbc-type="char" length="2"/>
                </field>
                <field name="zip" column="ZIP" indexed="true"/>
            </embedded>
        </field>
    </class>
</package>
<package name="org.mag.subscribe">
    <sequence name="ContractSeq" strategy="transactional"/>
    <class name="Contract">
        <inheritance strategy="subclass-table"/>
    </class>
    <class name="Subscription" table="CNTRCT.SUB">
        <inheritance>
            <discriminator value="1">
                <column name="TYPE" jdbc-type="tinyint"/>
            </discriminator>
        </inheritance>
        <version column="VERS" strategy="version-number"/>
        <field name="Contract.id" column="ID"/>
        <field name="startDate" column="START"/>
        <field name="payment" column="PAY"/>
        <field name="items">
            <element column="SUB_ID"/>
        </field>
    </class>
    <class name="LifetimeSubscription">
        <inheritance>
            <discriminator value="2"/>
        </inheritance>
        <field name="eliteClub" column="ELITE"/>
    </class>
</package>

```

```

<class name="TrialSubscription" table="CNTRCT.TRIAL_SUB">
  <inheritance strategy="new-table">
    <join column="ID" delete-action="restrict"/>
    <discriminator value="3"/>
  </inheritance>
  <field name="endDate" column="END"/>
</class>
<class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
  <version column="VERS" strategy="version-number"/>
  <field name="Contract.id" column="ID"/>
  <field name="price" column="PRICE"/>
  <field name="num" column="NUM" value-strategy="autoassign"/>
  <field name="magazine" delete-action="restrict">
    <column name="MAG_ISBN" target="ISBN"/>
    <column name="MAG_TITLE" target="TITLE"/>
  </field>
</class>
</package>
</orm>

```

Until now, we have focused on mapping metadata in the `.orm` file format. As we mentioned at the beginning of this chapter, however, you can integrate mapping metadata right into your `.jdo` file instead. Here are the above mappings integrated into our persistent class' JDO metadata:

Example 15.37. Integrated JDO Metadata Document

```

<?xml version="1.0"?>
<jdo>
  <package name="org.mag">
    <sequence name="ArticleSeq" datastore-sequence="ART_SEQ"/>
    <class name="Magazine" objectid-class="Magazine$MagazineId" table="MAG">
      <version column="VERS" strategy="version-number"/>
      <field name="isbn" primary-key="true">
        <column name="ISBN" jdbc-type="char" length="15"/>
      </field>
      <field name="title" column="TITLE" primary-key="true"/>
      <field name="price" column="PRICE"/>
      <field name="copiesSold" column="COPIES"/>
      <field name="coverArticle" column="COVER_ID" delete-action="restrict"/>
      <field name="articles" table="MAG_ARGS">
        <collection element-type="Article" dependent-element="true"/>
        <join delete-action="cascade">
          <column name="MAG_ISBN" target="ISBN"/>
          <column name="MAG_TITLE" target="TITLE"/>
        </join>
        <element column="ART_ID" delete-action="restrict"/>
      </field>
      <field name="publisher" column="PUB_ID" delete-action="restrict"/>
      <fetch-group name="detail">
        <field name="publisher" fetch-depth="0"/>
        <field name="articles" fetch-depth="0"/>
      </fetch-group>
    </class>
    <class name="Article" identity-type="application" detachable="true" table="ART">
      <version column="UPD" strategy="date-time"/>
      <field name="id" primary-key="true" column="ID" sequence="ArticleSeq"/>
      <field name="title" column="TITLE" unique="true"/>
      <field name="content" column="CONTENT"/>
      <field name="authors" table="ART_AUTHS">
        <map key-type="String" value-type="org.mag.pub.Author"/>
        <join column="ART_ID" delete-action="cascade"/>
        <key column="LNAME"/>
        <value column="AUTH_ID" delete-action="restrict"/>
      </field>
      <field name="subtitles" table="ART_SUBS">
        <collection element-type="String"/>
        <join column="ART_ID" delete-action="cascade"/>
        <element column="SUBTITLE"/>
        <order column="ORD"/>
      </field>
    </class>
  </package>
  <package name="org.mag.pub">
    <sequence name="AuthorSeq" factory-class="Author$SequenceFactory"/>
    <class name="Company" table="COMP">
      <datastore-identity column="CID" strategy="autoassign"/>
      <version strategy="none"/>
      <field name="name" column="NAME" unique="true"/>
      <field name="revenue" column="REV"/>
    </class>
  </package>

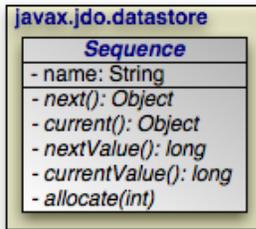
```

```

    <field name="address" embedded="true">
      <embedded null-indicator-column="STREET">
        <field name="street" column="STREET"/>
        <field name="city" column="CITY"/>
        <field name="state">
          <column name="STATE" jdbc-type="char" length="2"/>
        </field>
        <field name="zip" column="ZIP" indexed="true"/>
      </embedded>
    </field>
    <field name="subscriptions" table="COMP_SUBS">
      <collection element-type="org.mag.subscribe.Subscription"/>
      <join column="COMP_ID" delete-action="cascade"/>
      <element column="SUB_ID" delete-action="restrict"/>
    </field>
    <field name="mags" mapped-by="publisher">
      <collection element-type="org.mag.Magazine"/>
    </field>
  </class>
  <class name="Author" table="AUTH">
    <datastore-identity sequence="AuthorSeq">
      <column name="AID" sql-type="INTEGER64"/>
    </datastore-identity>
    <inheritance>
      <discriminator column="CLS" strategy="class-name"/>
    </inheritance>
    <version strategy="state-comparison"/>
    <field name="firstName" column="FNAME"/>
    <field name="lastName" column="LNAME" indexed="true"/>
    <field name="address" embedded="true">
      <embedded null-indicator-column="STREET">
        <field name="street" column="STREET"/>
        <field name="city" column="CITY"/>
        <field name="state">
          <column name="STATE" jdbc-type="char" length="2"/>
        </field>
        <field name="zip" column="ZIP" indexed="true"/>
      </embedded>
    </field>
  </class>
  <class name="Address" embedded-only="true"/>
</package>
<package name="org.mag.subscribe">
  <sequence name="ContractSeq" strategy="transactional"/>
  <class name="Contract" identity-type="application">
    <inheritance strategy="subclass-table"/>
    <field name="id" primary-key="true"/>
  </class>
  <class name="Subscription" detachable="true" table="CNTRCT.SUB">
    <inheritance>
      <discriminator value="1">
        <column name="TYPE" jdbc-type="tinyint"/>
      </discriminator>
    </inheritance>
    <version column="VERS" strategy="version-number"/>
    <field name="Contract.id" column="ID"/>
    <field name="startDate" column="START"/>
    <field name="payment" column="PAY"/>
    <field name="items">
      <collection element-type="Subscription$LineItem" dependent-element="true"/>
      <element column="SUB_ID"/>
      <extension vendor-name="kodo" key="lock-group" value="none"/>
    </field>
  </class>
  <class name="LifetimeSubscription">
    <inheritance>
      <discriminator value="2"/>
    </inheritance>
    <field name="eliteClub" column="ELITE"/>
  </class>
  <class name="TrialSubscription" table="CNTRCT.TRIAL_SUB">
    <inheritance strategy="new-table">
      <join column="ID" delete-action="restrict"/>
      <discriminator value="3"/>
    </inheritance>
    <field name="endDate" column="END"/>
  </class>
  <class name="Subscription$LineItem" table="CNTRCT.LINE_ITEM">
    <version column="VERS" strategy="version-number"/>
    <field name="Contract.id" column="ID"/>
    <field name="price" column="PRICE"/>
    <field name="num" column="NUM" value-strategy="autoassign"/>
    <field name="magazine" delete-action="restrict">
      <column name="MAG_ISBN" target="ISBN"/>
      <column name="MAG_TITLE" target="TITLE"/>
    </field>
  </class>
</package>
</jdo>

```

Chapter 16. Sequence



Sequences are simple value generators. As we saw in **Chapter 15, Mapping Metadata [260]**, sequences are typically used to automatically create identity or field values for persistent objects. At times, however, you may want to use a sequence directly through JDO's Sequence interface.

```
public Sequence getSequence (String name);
```

Chapter 15, Mapping Metadata [260] demonstrated how to define a named sequence in metadata. The `PersistenceManager`'s `getSequence` method retrieves a sequence by its fully qualified name. If no sequence with the given name exists, the `PersistenceManager` throws a `JDOUserException`.

Note

In Kodo, you can also obtain the identity sequence of a class or the value sequence of a field through the `kodo.jdo.KodoJDOHelper`.

```
public Object next ();
public Object current ();
public long nextValue ();
public long currentValue ();
```

The `Sequence.next` method returns the next value in the sequence. The type of the return value will depend on the sequence type, as defined in metadata. In JDOR, most JDO sequences are front-ends for native database sequences, which have integral return types, such as `Long`.

Invoking `current` returns the current sequence value, which is also typically the last sequence value used. Some sequences cannot access their current value without incrementing it, or do not track their current value. Calling `current` on these sequences returns `null`.

`nextValue` and `currentValue` are convenience methods for sequences that return integral values. They are equivalent to invoking:

```
((Long) next ().longValue ())
((Long) current ().longValue ())
```

```
public void allocate (int additional);
```

This method is a hint that you will be requesting additional sequence values in the near future. The implementation might optimize by generating the needed sequence values in batch.

Example 16.1. Using Sequences

```
PersistenceManager pm = ...;
Sequence seq = pm.getSequence ("org.mag.subscribe.ContractSeq");
seq.allocate (1000); // hint for efficiency
for (int i = 0; i < 1000; i++)
{
    long contractId = seq.nextValue ();
    pm.makePersistent (generateContract (contractId));
}
```

Chapter 17. SQL Queries

JDOQL is a powerful query language, but there are times when it is not enough. Maybe you're migrating a JDBC application to JDO on a strict deadline, and you don't have time to translate your existing SQL selects to JDOQL. Or maybe a certain query requires database-specific SQL your JDO implementation doesn't support. Or maybe your DBA has spent hours crafting the perfect select statement for a query in your application's critical path. Whatever the reason, SQL queries can remain an essential part of an application.

You are probably familiar with executing SQL queries by obtaining a `java.sql.Connection`, using the JDBC APIs to create a `Statement`, and executing that `Statement` to obtain a `ResultSet`. And of course, you are free to continue using this low-level approach to SQL execution in your JDO applications. However, JDO also supports executing SQL queries through the `javax.jdo.Query` interface introduced in [Chapter 11, Query \[232\]](#). Using a JDO SQL query, you can retrieve either persistent objects or projections of column values. The following sections detail each use.

Important

This feature requires **Kodo Enterprise Edition**.

Note

Kodo also supports embedding SQL into standard JDOQL queries. See [Section 9.6, “Query Language Extensions” \[?\]](#) in the Reference Guide for details.

17.1. Creating SQL Queries

The `PersistenceManager` has two factory methods suitable for creating SQL queries:

```
public Query newQuery (Object query);
public Query newQuery (String language, Object query);
```

The first method is used to create a new `Query` instance with the same properties as the passed-in template. The template might be a `Query` from another `PersistenceManager`, or a `Query` that has been deserialized and has lost its `PersistenceManager` association. The method works for any query, regardless of the language used.

The second method was designed specifically for non-JDOQL queries. Its first parameter is the query language to use. For SQL queries, the language is `javax.jdo.query.SQL` (in case you are wondering, the official JDOQL language string is `javax.jdo.query.JDOQL`). Its second parameter represents the query to run - in this case, the SQL string. The example below shows these methods in action.

Example 17.1. Creating a SQL Query

```
PersistenceManager pm = ...;
Query query = pm.newQuery ("javax.jdo.query.SQL", "SELECT * FROM MAG");
query.setClass (Magazine.class);
processMagazines ((List) query.execute ());
query.closeAll ();

Query template = deserializeTemplateQuery ();
query = pm.newQuery (template);
processMagazines ((List) query.execute ());
query.closeAll ();
```

While JDOQL queries have separate result, filter, grouping, and ordering strings, a single `SELECT` statement encompasses a complete SQL query. Thus, most methods of `SQL Query` objects throw an exception. In particular, you cannot call the following methods:

- `setCandidates (Collection)`
- `setFilter (String)`
- `setResult (String)`
- `setGrouping (String)`
- `setOrdering (String)`
- `declareImports (String)`
- `declareVariables (String)`
- `declareParameters (String)`

Note

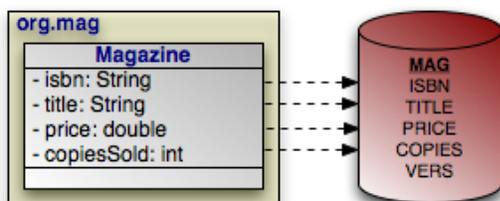
In addition to `SELECT` statements, Kodo supports stored procedure invocations as SQL queries. Kodo will assume any SQL that does not begin with the `SELECT` keyword (ignoring case) is a stored procedure call, and invoke it as such at the JDBC level.

17.2. Retrieving Persistent Objects with SQL

When you give a `SQL Query` a candidate class, it will return persistent instances of that class. At a minimum, your SQL must select the class' primary key columns, discriminator column (if mapped), and version column (also if mapped). The JDO runtime uses the values of the primary key columns to construct each result object's identity, and possibly to match it with a persistent object already in the `PersistenceManager`'s cache. When an object is not already cached, the implementation creates a new object to represent the current result row. It might use the discriminator column value to make sure it constructs an object of the correct subclass. Finally, the query records available version column data for use in optimistic concurrency checking, should you later change the result object and flush it back to the database.

Aside from the primary key, discriminator, and version columns, any columns you select are used to populate the persistent fields of each result object. JDO implementations will compete on how effectively they map your selected data to your persistent instance fields.

Let's make the discussion above concrete with an example. It uses the following simple mapping between a class and the database:



Example 17.2. Retrieving Persistent Objects

```

Query query = pm.newQuery ("javax.jdo.query.SQL", "SELECT ISBN, TITLE, PRICE, "
    + "VERS FROM MAG WHERE PRICE > 5 AND PRICE < 10");
query.setClass (Magazine.class);
List results = (List) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ());
    processMagazine ((Magazine) itr.next ());
query.close (results);

```

It is very important to notice that we explicitly set the candidate class for the query. If we had not performed this step, the query would have been treated as a projection. We cover SQL projections later in this chapter.

The query above works as advertised, but isn't very flexible. Let's update it to take in parameters for the minimum and maximum price, so we can reuse it to find magazines in any price range:

Example 17.3. SQL Query Parameters

```

Query query = pm.newQuery ("javax.jdo.query.SQL", "SELECT ISBN, TITLE, PRICE, "
    + "VERS FROM MAG WHERE PRICE > ? AND PRICE < ?");
query.setClass (Magazine.class);

Double min = new Double (5D);
Double max = new Double (10D);
List results = (List) query.execute (min, max);
for (Iterator itr = results.iterator (); itr.hasNext ());
    processMagazine ((Magazine) itr.next ());
query.close (results);

```

Like JDBC prepared statements, SQL queries represent parameters with question marks. When you run a query with multiple parameters, the order of your arguments to the `execute` method must match the order of the question marks for each parameter in your SQL. To use the Query interface's `executeWithMap` method, imagine that the question marks are labeled from 1 to N, and key each parameter value on the correct Integer position. Here is the above example again, this time using a parameter map:

Example 17.4. SQL Query Parameter Map

```

Query query = pm.newQuery ("javax.jdo.query.SQL", "SELECT ISBN, TITLE, PRICE, "
    + "VERS FROM MAG WHERE PRICE > ? AND PRICE < ?");
query.setClass (Magazine.class);

Map params = new HashMap ();
params.put (new Integer (1), new Double (5D));
params.put (new Integer (2), new Double (10D));
List results = (List) query.executeWithMap (params);
for (Iterator itr = results.iterator (); itr.hasNext ());
    processMagazine ((Magazine) itr.next ());
query.close (results);

```

17.3. SQL Projections

SQL queries without a candidate class are treated as projections of column data. If you select a single column, the query returns a

list of Objects. If you select multiple columns, it returns a list of `Object []`s. In either case, each column value is obtained using the `java.sql.ResultSet.getObject` method. The following example demonstrates a query for the values of the ISBN and VERS columns of all MAG table records, using the data model we defined in [Section 17.2, “Retrieving Persistent Objects with SQL” \[321\]](#).

Example 17.5. Column Projection

```
Query query = pm.newQuery ("javax.jdo.query.SQL",
    "SELECT ISBN, VERS FROM MAG");
List results = (List) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
{
    Object[] data = (Object[]) results.next ();
    processISBNAndVersion (data[0], data[1]);
}
query.close (results);
```

Notice that in the code above, we did not set a candidate class. Therefore, the query is treated as a projection.

Our discussion of JDOQL query result classes in [Section 11.8, “Result Class” \[246\]](#) also applies to SQL queries. As with JDOQL queries, SQL queries can automatically pack their results into objects of a specified type. JDO uses the `java.sql.ResultSetMetaData.getColumnLabel` method to match each column alias to the result class' public fields and JavaBean setter methods. Here is a modification of our example above that packs the selected column values into JavaBean instances.

Example 17.6. Result Class

```
public class Identity
{
    private String id;
    private int versionNumber;

    public void setId (String id)
    {
        this.id = id;
    }

    public String getId ()
    {
        return id;
    }

    public void setVersionNumber (int versionNumber)
    {
        this.versionNumber = versionNumber;
    }

    public int getVersionNumber ()
    {
        return versionNumber;
    }
}

Query query = pm.newQuery ("javax.jdo.query.SQL",
    "SELECT ISBN AS id, VERS AS versionNumber FROM MAG");
query.setResultClass (Identity.class);
List results = (List) query.execute ();
for (Iterator itr = results.iterator (); itr.hasNext ();)
    processIdentity ((Identity) itr.next ());
query.close (results);
```

17.4. Named SQL Queries

We discussed how to write named JDOQL queries in [Section 11.10, “Named Queries” \[250\]](#). Named queries, however, are not limited to JDOQL. By setting the `query` element's `language` attribute to `javax.jdo.query.SQL`, you can define a named SQL query. A named SQL query within a `class` element queries for instances of that class; a named SQL query outside of a `class` element acts as a column data projection.

Example 17.7. Named SQL Queries

```
<?xml version="1.0"?>
<jdoquery>
  <query name="salesReport" language="javax.jdo.query.SQL">
    SELECT TITLE, PRICE * COPIES FROM MAG
  </query>
  <package name="org.mag">
    <class name="Magazine">
      <query name="findByTitle" language="javax.jdo.query.SQL">
        SELECT * FROM MAG WHERE TITLE = ?
      </query>
    </class>
  </package>
</jdoquery>
```

The `salesReport` query above returns the title and revenue generated for each Magazine. Because it is a projection, it does not have a candidate class, and so we specify it at the root level.

The `findByTitle` query returns the Magazine with the title given on execution. The code below executes both queries.

```
PersistenceManager pm = ...;
Query query = pm.newNamedQuery (null, "salesReport");
List sales = (List) query.execute ();
for (Iterator itr = sales.iterator (); itr.hasNext ();)
{
  Object[] salesData = (Object[]) itr.next ();
  processSalesData ((String) salesData[0], (Number) salesData[1]);
}
query.close (sales);

query = pm.newNamedQuery (Magazine.class, "findByTitle");
query.setUnique (true);
Magazine jdj = (Magazine) query.execute ("JDJ");
```

17.5. Conclusion

If you've used relational databases extensively, you might be tempted to perform all your JDO queries with SQL. Try to resist this temptation. SQL queries tie your application to the particulars of your current table model and database vendor. If you stick with JDOQL, on the other hand, you can port your application to other schemas and database vendors without any changes to your code. Additionally, most JDO implementations already produce highly optimized SQL from your JDOQL filters, and many are able to cache JDOQL query results for added performance.

Chapter 18. Conclusion

This concludes our overview of the Java Data Objects specification. The **Kodo JPA/JDO Tutorials** continue your JDO education with step-by-step instructions for building simple JDO applications. Next, the **Kodo JPA/JDO Reference Guide** contains detailed documentation on all aspects of Kodo' implementation and core development tools.

Part 4. Tutorials

Table of Contents

1. JPA Tutorials	329
1.1. Kodo JPA Tutorials	329
1.1.1. Tutorial Requirements	329
1.2. Kodo JPA Tutorial	329
1.2.1. The Pet Shop	329
1.2.1.1. Included Files	329
1.2.1.2. Important Utilities	330
1.2.2. Getting Started	330
1.2.2.1. Configuring the Datastore	332
1.2.3. Inventory Maintenance	333
1.2.3.1. Persisting Objects	334
1.2.3.2. Deleting Objects	335
1.2.4. Inventory Growth	336
1.2.5. Behavioral Analysis	338
1.2.5.1. Complex Queries	341
1.2.6. Extra Features	342
1.3. J2EE Tutorial	342
1.3.1. Prerequisites for the Kodo J2EE Tutorial	342
1.3.2. J2EE Installation Types	342
1.3.3. Installing Kodo Into Pre-J2EE 5 Application Servers	343
1.3.4. Installing the J2EE Sample Application	343
1.3.4.1. Compiling and Building The Sample Application	343
1.3.4.2. Deploying Sample To JBoss	344
1.3.4.3. Deploying Sample To WebLogic 9	344
1.3.5. Using The Sample Application	344
1.3.6. Sample Architecture	344
1.3.7. Code Notes and J2EE Tips	345
2. JDO Tutorials	346
2.1. Kodo JDO Tutorials	346
2.1.1. Tutorial Requirements	346
2.2. Kodo JDO Tutorial	346
2.2.1. The Pet Shop	346
2.2.1.1. Included Files	346
2.2.1.2. Important Utilities	347
2.2.2. Getting Started	347
2.2.2.1. Configuring the Datastore	348
2.2.3. Inventory Maintenance	349
2.2.3.1. Persisting Objects	350
2.2.3.2. Deleting Objects	351
2.2.4. Inventory Growth	352
2.2.5. Behavioral Analysis	354
2.2.5.1. Complex Queries	357
2.2.6. Extra Features	358
2.3. Reverse Mapping Tool Tutorial	358
2.3.1. Magazine Shop	358
2.3.2. Setup	359
2.3.2.1. Tutorial Files	359
2.3.2.2. Important Utilities	359
2.3.3. Generating Persistent Classes	359
2.3.4. Using the Finder	361
2.4. J2EE Tutorial	362
2.4.1. Prerequisites for the Kodo J2EE Tutorial	362
2.4.2. J2EE Installation Types	363

2.4.3. Installing Kodo	363
2.4.4. Installing the J2EE Sample Application	363
2.4.4.1. Compiling and Building The Sample Application	363
2.4.4.2. Deploying Sample To JBoss	364
2.4.4.3. Deploying Sample To WebLogic 6.1 to 7.x	364
2.4.4.4. Deploying Sample To WebLogic 8.1	364
2.4.4.5. Deploying Sample To SunONE / Sun JES	365
2.4.4.6. Deploying Sample To JRun	365
2.4.4.7. Deploying Sample To WebSphere	365
2.4.4.8. Deploying Sample To Borland Enterprise Server 5.2	365
2.4.5. Using The Sample Application	365
2.4.6. Sample Architecture	365
2.4.7. Code Notes and J2EE Tips	366

Chapter 1. JPA Tutorials

1.1. Kodo JPA Tutorials

These tutorials provide step-by-step examples of how to use various facets of the Kodo JPA system. They assume a general knowledge of JPA and Java. For more information on these subjects, see the following URLs:

- [Sun's Java site](#)
- [JPA Overview Document](#)
- [Links to JPA](#)

1.1.1. Tutorial Requirements

These tutorials require that JDK 1.5 or greater be installed on your computer, and that `java` and `javac` are in your `PATH` when you open a command shell. See [Chapter 2, *Kodo Installation*](#) [4] of Part I of this manual (the Kodo README) or see the [README.txt](#) included in the root directory of your download for more information on requirements and installation procedures.

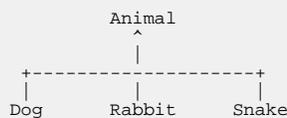
1.2. Kodo JPA Tutorial

In this tutorial you will become familiar with the basic tools and development processes under Kodo by creating a simple JPA application.

1.2.1. The Pet Shop

Imagine that you have decided to create a software toolkit to be used by pet shop operators. This toolkit must provide a number of solutions to common problems encountered at pet shops. Industry analysts indicate that the three most desired features are inventory maintenance, inventory growth simulation, and behavioral analysis. Not one to question the sage advice of experts, you choose to attack these three problems first.

According to the aforementioned experts, most pet shops focus on three types of animals only: dogs, rabbits, and snakes. This ontology suggests the following class hierarchy:



1.2.1.1. Included Files

We have provided an implementation of `Animal` and `Dog` classes, plus some helper classes and files to create the initial schema and populate the database with some sample dogs. Let's take a closer look at these classes.

- **`tutorial.persistence.AnimalMaintenance`**: Provides some utility methods for examining and manipulating the animals stored in the database. We will fill in method definitions in [Section 1.2.3, "Inventory Maintenance"](#) [333]

- `tutorial.persistence.Animal`: This is the superclass of all animals that this pet store software can handle.
- `tutorial.persistence.Dog`: Contains data and methods specific to dogs.
- `tutorial.persistence.Rabbit`: Contains data and methods specific to rabbits. It will be used in [Section 1.2.4, “Inventory Growth” \[336\]](#)
- `tutorial.persistence.Snake`: Contains data and methods specific to snakes. It will be used in [Section 1.2.5, “Behavioral Analysis” \[338\]](#)
- `../META-INF/persistence.xml`: This XML file contains Kodo-specific and standard JPA configuration settings.

Important

You must ensure you have valid BEA license key available to Kodo. For further details on Kodo licensing, see [Section 2.2, “License Configuration” \[393\]](#)

It is important to load all persistent entity classes at startup so that Kodo can match database discriminator values to entity classes. Often this happens automatically. Some parts of this tutorial, however, do require that all entity classes be loaded explicitly. The JPA standard includes persistent class listings in its XML configuration format. Add the following lines to `../META-INF/persistence.xml` between the `<provider>` and the `<properties>` elements:

```
<class>tutorial.persistence.Animal</class>
<class>tutorial.persistence.Dog</class>
```

- **solutions**: The solutions directory contains the complete solutions to this tutorial, including finished versions of the `.java` files listed above.

1.2.1.2. Important Utilities

- **java**: Runs main methods in specified Java classes.
- **javac**: Compiles `.java` files into `.class` files that can be executed by **java**.
- **kodoc**: Runs the Kodo enhancer against the specified classes. More information is available in [Section 5.2, “Enhancement” \[452\]](#) of the Reference Guide.
- **mappingtool**: A utility that can be used to create and maintain the object-relational mappings and schema of all persistent classes in a JDBC-compliant datastore. This functionality allows the underlying mappings and schema to be easily kept up-to-date with the Java classes in the system. See [Chapter 7, Mapping \[489\]](#) of the Reference Guide for more information.

1.2.2. Getting Started

Let's compile the initial classes and see them in action. To do so, we must compile the `.java` files, as we would with any Java project, and then pass the resulting classes through the Kodo enhancer:

Note

Be sure that your `CLASSPATH` is set correctly. Please see [Section 2.2.7, “Windows Installation” \[5\]](#) of Part I of this manual for Windows, or [Section 2.2.8, “POSIX \(Linux, Solaris, Mac OS X, Windows with cygwin, etc.\) Installation” \[6\]](#) for other operating systems.

tion” [5] of Part I of this manual for POSIX (Linux, Solaris, Cygwin, etc.). Detailed information on which libraries are needed can be found in [Appendix 7, Development and Runtime Libraries \[651\]](#). Note, also, that your Kodo install directory should be in the CLASSPATH, as the tutorial classes are located in the `tutorial/persistence` directory under your Kodo install directory, and are in the `tutorial.persistence` package.

1. Make sure you are in the `tutorial/persistence` directory. All examples throughout the tutorial assume that you are in this directory.
2. Examine `Animal.java`, `Dog.java`, and `SeedDatabase.java`

These files are good examples of the simplicity JPA engenders. As noted earlier, persisting an object or manipulating an object's persistent data requires almost no JPA-specific code. For a very simple example of creating persistent objects, please see the `seed` method of `SeedDatabase.java`. Note the objects are created with normal Java constructors. The files `Animal.java` and `Dog.java` are also good examples of how JPA allows you to manipulate persistent data without writing any specific JPA code, by providing simple annotations.

Let's take a look at the `Animal.java` file. Notice that the class is a Plain Old Java Object (POJO), with several annotations describing how the class is mapped into a relational database. First, let's examine the class level annotations:

```
@Entity(name="Animal")
@Table(name="JPA_TUT_ANIMAL")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="SPECIES", length=100)
public abstract class Animal
{
    ...
}
```

The annotations serve to map the class into the database. For more information on these and other annotations, see [Chapter 5, Metadata \[28\]](#) and [Chapter 12, Mapping Metadata \[118\]](#).

- a. **@Entity**: This annotation indicates that instances of this class may be persistent entities. The value of the **name** attribute is the entity name, and is used in queries, etc.
- b. **@Table**: This annotation is used to map the entity to a primary table. The value of the **name** attribute specifies the name of the relational table to use as the primary table.
- c. **@Inheritance**: When multiple classes in an inheritance hierarchy are persistent entity types, it is important to describe how the inheritance hierarchy is mapped. Setting the value of the **strategy** attribute to **InheritanceType.SINGLE_TABLE** indicates that the primary table for all subclasses shall be the same table as for the superclass.
- d. **@DiscriminatorColumn**: With a **SINGLE_TABLE** inheritance mapping strategy, instances of multiple classes will be stored in the same table. This annotation describes a column in that table that is used to determine the type of an instance whose data is stored in a particular row. The **name** attribute is the name of the column, and the **length** attribute indicates the size of the column. By default, the unqualified class name for the instance is stored in the discriminator column. To store a different value for a type, use the **@DiscriminatorValue** annotation.

Let's take a look at our class' field annotations. We have chosen to use *field access* for our entities, meaning the persistence implementation will get and set persistent state directly through our class' declared fields. We could have chosen to use *property access*, in which the implementation accesses persistent state through our JavaBean getter and setter methods. In that case, we would have annotated our getter methods rather than our fields.

```
@Id
@GeneratedValue
@Column(name="ID")
private long id;

@Basic @Column(name="ANIMAL_NAME")
private String name = null;
```

```
@Basic @Column(name="COST")
private float price = 0f;
```

The annotations serve to map the fields into the database. For more information on these and other annotations, see [Chapter 5, Metadata \[28\]](#).

- a. **@Id**: This annotation indicates that the field is to be mapped to a primary key column in the database.
 - b. **@GeneratedValue**: Indicates that the implementation will generate a value for the field automatically.
 - c. **@Column**: This annotation describes the column to which the field will be mapped. The **name** attribute specifies the name of the column.
 - d. **@Basic**: This annotation indicates that the field is simply mapped into a column. There are other annotations that indicate entity relationships and other more complex mappings.
3. Compile the `.java` files.

```
javac *.java
```

You can use any java compiler instead of **javac**.

4. Enhance the persistent classes.

```
kodoc -p persistence.xml Animal.java Dog.java
```

This step runs the Kodo enhancer on the `Animal.java` and `Dog.java` files mentioned above. See [Section 5.2, “Enhancement” \[452\]](#) of the Reference Guide for more information on the enhancer, including how to use automatic runtime enhancement.

Note

The `-p` flag points the enhancer to your `persistence.xml` configuration file. All Kodo JPA tools look for default configuration in a resource called `kodo.xml` or `META-INF/kodo.xml`. Thus you can avoid passing the `-p` argument to tools by using this configuration file name in place of `persistence.xml`. See [Chapter 2, Configuration \[?\]](#) in the Reference Guide for details on Kodo configuration.

1.2.2.1. Configuring the Datastore

Now that we've compiled the source files and enhanced the persistent classes, we're ready to set up the database. **Hypersonic SQL**, a pure Java relational database, is included in the Kodo distribution. We have included this database because it is simple to set up and has a small memory footprint; however, you can use this tutorial with any of the relational databases that we support. You can also write your own plugin for any database that we do not support. For the sake of simplicity, this tutorial only describes how to set up connectivity to a Hypersonic SQL database. For more information on how to connect to a different database or how to add support for other databases, see [Chapter 4, JDBC \[423\]](#) of the Reference Guide.

1. Create the object-relational mappings and database schema.

```
mappingtool -p persistence.xml Animal.java Dog.java
```

This command propagates the necessary schema for the specified classes to the database configured in `persistence.xml`. If you are using the default Hypersonic SQL setup, the first time you run the mapping tool Hypersonic will create `tutorial_database.properties` and `tutorial_database.script` database files in your current directory. To delete the database, just delete these files.

By default, JPA uses object-relational mapping information stored in annotations in your source files. **Chapter 12, Mapping Metadata [118]** of the JPA Overview will help you understand mapping annotations. Additionally, **Chapter 7, Mapping [489]** of the Reference Guide describes your other mapping options in detail.

If you are curious, you can view the schema Kodo created for the tutorial classes with Kodo's schema tool:

```
schematool -p persistence.xml -a reflect -f tmp.schema
```

This will create a `tmp.schema` file with an XML representation of the database schema. The XML should be self explanatory; see **Section 4.14, “XML Schema Format” [447]** of the Reference Guide for details. You may delete the `tmp.schema` file before proceeding.

2. Populate the database with sample data.

```
java tutorial.persistence.SeedDatabase
```

Congratulations! You have now created an JPA-accessible persistent store, and seeded it with some sample data.

1.2.3. Inventory Maintenance

The most important element of a successful pet store product, say the experts, is an inventory maintenance mechanism. So, let's work on the `Animal` and `Dog` classes a bit to permit user interaction with the database.

This chapter should familiarize you with some of the basics of the **JPA specification** and the mechanics of compiling and enhancing persistent classes. You will also become familiar with the mapping tool for propagating the persistent schema into the database.

First, let's add some code to `AnimalMaintenance.java` that allows us to examine the animals currently in the database.

1. Add code to `AnimalMaintenance.java`.

Modify the `getAnimals` method of `AnimalMaintenance.java` to look like this:

```
/**
 * Return a list of animals that match the specified query filter.
 *
 * @param filter the JPQL filter to apply to the query
 * @param cls the class of animal to query on
 * @param em the EntityManager to obtain the query from
 */
```

```

    */
    public static List getAnimals (String filter, EntityManager em)
    {
        // Execute a query for the specified filter.
        Query query = em.createQuery (filter);
        return query.getResultList ();
    }

```

2. Compile `AnimalMaintenance.java`.

```
javac AnimalMaintenance.java
```

3. Take a look at the animals in the database.

```
java tutorial.persistence.AnimalMaintenance list Animal
```

Notice that `list` optionally takes a query filter. Let's explore the database some more, this time using filters:

```

java tutorial.persistence.AnimalMaintenance list "select a from Animal a where a.name = 'Binney'"
java tutorial.persistence.AnimalMaintenance list "select a from Animal a where a.price <= 50"

```

The Java Persistence Query Language (JPQL) is designed to look and behave much like an object oriented SQL dialect. The `name` and `price` fields identified in the above queries map to the member fields of those names in `tutorial.persistence.Animal`. More details on JPQL syntax is available in **Chapter 10, JPA Query [82]** of the JPA Overview.

Great! Now that we can see the contents of the database, let's add some code that lets us add and remove animals.

1.2.3.1. Persisting Objects

As new dogs are born or acquired, the store owner will need to add new records to the inventory database. In this section, we'll write the code to handle additions through the `tutorial.persistence.AnimalMaintenance` class.

This section will familiarize you with the mechanism for storing persistent instances in a JPA entity manager. We will create a new dog, obtain a `Transaction` from a `EntityManager`, and, within the transaction, make the new dog object persistent.

`tutorial.persistence.AnimalMaintenance` provides a reflection-based facility for creating any type of animal, provided that the animal has a two-argument constructor whose first argument corresponds to the name of the animal and whose second argument is an implementation-specific primitive. This reflection-based system is in place to keep this tutorial short and remove repetitive creation mechanisms. It is not a required part of the JPA specification.

1. Add the following code to `AnimalMaintenance.java`.

Modify the `persistObject` method of `AnimalMaintenance.java` to look like this:

```
/**
```

```

* Performs the actual JPA work of putting <code>object</code>
* into the data store.
*
* @param object the object to persist in the data store
*/
public static void persistObject (EntityManager em, Object object)
{
    // Mark the beginning of the unit of work boundary.
    em.getTransaction ().begin ();

    em.persist (object);

    // Mark the end of the unit of work boundary,
    // and record all inserts in the database.
    em.getTransaction ().commit ();
    System.out.println ("Added " + object);
}

```

Note

In the above code, we pass in an `EntityManager`. `EntityManagers` may be either container managed or application managed. In this tutorial, because we're operating outside a container, we're using application managed `EntityManagers`. In managed environments, `EntityManagers` are typically container managed, and thus injected or looked up via JNDI. Application managed `EntityManagers` can be used in both managed and unmanaged environments, and are created by an `EntityManagerFactory`. An `EntityManagerFactory` can be obtained from the `javax.persistence.Persistence` class. This class provides some convenience methods for obtaining an `EntityManagerFactory`.

2. Recompile `AnimalMaintenance.java`.

```
javac AnimalMaintenance.java
```

You now have a mechanism for adding new dogs to the database. Go ahead and add some by running **java tutorial.persistence.AnimalMaintenance add Dog <name> <price>** For example:

```
java tutorial.persistence.AnimalMaintenance add Dog Fluffy 35
```

You can view the contents of the database with:

```
java tutorial.persistence.AnimalMaintenance list Dog
```

1.2.3.2. Deleting Objects

What if someone decides to buy one of the dogs? The store owner will need to remove that animal from the database, since it is no longer in the inventory.

This section demonstrates how to remove data from the datastore.

1. Add the following code to `AnimalMaintenance.java` .

Modify the `deleteObjects` method of `AnimalMaintenance.java` to look like this:

```
/**
 * Performs the actual JPA work of removing
 * <code>objects</code> from the datastore.
 *
 * @param objects    the objects to persist in the datastore
 * @param em         the EntityManager to delete with
 */
public static void deleteObjects (Collection objects, EntityManager em)
{
    // Mark the beginning of the unit of work boundary.
    em.getTransaction ().begin ();

    // This method removes the objects in 'objects' from the data store.
    for (Object ob : objects)
    {
        System.out.println ("Removed animal: " + ob);
        em.remove (ob);
    }

    // Mark the end of the unit of work boundary, and record all
    // deletes in the database.
    em.getTransaction ().commit ();
}
```

2. Recompile `AnimalMaintenance.java`.

```
javac AnimalMaintenance.java
```

3. Remove some animals from the database.

```
java tutorial.persistence.AnimalMaintenance remove <query>
```

Where `<query>` is a query string like those used for listing animals above.

All right. We now have a basic pet shop inventory management system. From this base, we will add some of the more advanced features suggested by our industry experts.

1.2.4. Inventory Growth

Now that we have the basic pet store framework in place, let's add support for the next pet in our list: the rabbit. The rabbit is a bit different than the dog; pet stores sell them all for the same price, but gender is critically important since rabbits reproduce rather easily and quickly. Let's put together a class representing a rabbit.

In this chapter, you will see some more queries and write a bidirectional relation between objects.

Provided with this tutorial is a file called `Rabbit.java` which contains a sample `Rabbit` implementation.

1. Examine `Rabbit.java`.
2. The `Rabbit` class above contains a bidirectional relationship between parents and children. From the Java side of things, a bidirectional relationship is simply a pair of fields that are conceptually linked. There is no special Java work necessary to ex-

press bidirectionality. However, you must identify the relationship as bidirectional using JPA **annotations** so the mapping tool can create the most efficient schema.

Insert this snippet of code immediately *before* the `children` field declaration in the `Rabbit.java` file.

```
@ManyToMany
@JoinTable(name="RABBIT_CHILDREN",
    joinColumns=@JoinColumn(name="PARENT_ID"),
    inverseJoinColumns=@JoinColumn(name="CHILD_ID"))
```

The `@ManyToMany` annotation indicates that `children` is one side of a many-to-many relation. `@JoinTable` describes how this relation maps to a database join table. The annotation's `joinColumns` name the join table's foreign key columns linking to the owning instance (the parent). In this case, column `RABBIT_CHILDREN.PARENT_ID` is a foreign key to the parent's ID primary key column. Similarly, the `inverseJoinColumns` attribute denotes the foreign key columns linking to the collection elements (the children). For more details on the `@JoinTable` annotation, see [Chapter 12, Mapping Metadata \[118\]](#) of the JPA Overview.

Now we'll map the other side of this bidirectional relation, the `parents` field. Insert the following snippet of code immediately *before* the `parents` field declaration in the `Rabbit.java` file. The `mappedBy` attribute identifies the name of the owning side of the relation.

```
@ManyToMany(mappedBy="children")
```

3. Compile `Rabbit.java`.

```
javac Rabbit.java
```

4. Enhance the `Rabbit` class.

```
kodoc -p persistence.xml Rabbit.java
```

5. Refresh the object-relational mappings and database schema.

```
mappingtool -p persistence.xml Rabbit.java
```

Now that we have a `Rabbit` class, let's get some preliminary rabbit data into the database.

1. Add a `<class>` entry for `Rabbit` to `../../META-INF/persistence.xml`.

```
<class>tutorial.persistence.Rabbit</class>
```

2. Create some rabbits.

Run the following commands a few times to add some male and female rabbits to the database:

```
java tutorial.persistence.AnimalMaintenance add Rabbit <name> false
java tutorial.persistence.AnimalMaintenance add Rabbit <name> true
```

Now run some breeding iterations.

```
java tutorial.persistence.Rabbit breed 2
```

3. Look at your new rabbits.

```
java tutorial.persistence.AnimalMaintenance list Rabbit
java tutorial.persistence.AnimalMaintenance details "select r from Rabbit r where r.name = '<name>'"
```

1.2.5. Behavioral Analysis

Often, pet stores sell snakes as well as rabbits and dogs. Pet stores are primarily concerned with a snake's length; much like rabbits, pet store operators usually sell them all for a flat rate.

This chapter demonstrates more queries, schema manipulation, and additional relation types.

Provided with this tutorial is a file called `Snake.java` which contains a sample Snake implementation. Let's get it compiled and loaded:

1. Examine and compile `Snake.java`.

```
javac Snake.java
```

2. Enhance the class.

```
kodoc -p persistence.xml Snake.java
```

3. Refresh the mappings and database.

As we have created a new persistent class, we must map it to the database and change the schema to match. So run the mapping tool:

```
mappingtool -p persistence.xml Snake.java
```

4. Add a `<class>` entry for Snake to `../META-INF/persistence.xml`.

```
<class>tutorial.persistence.Snake</class>
```

Once you have compiled everything, add a few snakes to the database using:

```
java tutorial.persistence.AnimalMaintenance add Snake "name" <length>
```

Where `<length>` is the length in feet for the new snake. To see the new snakes in the database, run:

```
java tutorial.persistence.AnimalMaintenance list Snake
```

Unfortunately for the massively developing rabbit population, snakes often eat rabbits. Any good inventory system should be able to capture this behavior. So, let's add some code to `Snake.java` to support the snake's eating behavior.

First, let's modify `Snake.java` to contain a list of eaten rabbits.

1. Add the following code snippets to `Snake.java`.

```
// This list will be persisted into the database as
// a one-to-many relation.
@OneToMany(mappedBy="eater")
private Set<Rabbit> giTract = new HashSet<Rabbit> ();
```

Note that we specified a `mappedBy` attribute in this example. This is because the relation is bidirectional; that is, the rabbit has knowledge of which snake ate it. We could have left out the `eater` field and instead created a standard unidirectional relation. In fact, in a bidirectional many-to-one relation, the many side must always be the owner.

For more information on types of relations, see [Section 12.8, “Field Mapping” \[143\]](#) of the JPA Overview.

Modify the `toString (boolean)` method to output the `giTract` list.

```
public String toString (boolean detailed)
{
    StringBuffer buf = new StringBuffer (1024);
    buf.append ("Snake ").append (getName ());

    if (detailed)
    {
        buf.append (" (").append (length).append (" feet long) sells for ");
        buf.append (getPrice ()).append (" dollars.");
        buf.append (" Its gastrointestinal tract contains:\n");
        for (Rabbit rabbit : giTract)
            buf.append ("\t").append (rabbit).append ("\n");
    }
    else
        buf.append ("; ate " + giTract.size () + " rabbits.");

    return buf.toString ();
}
```

Add the following methods.

```

/**
 * Kills the specified rabbit and eats it.
 */
public void eat (Rabbit dinner)
{
    // Consume the rabbit.
    dinner.kill ();
    dinner.setEater (this);
    giTract.add (dinner);
    System.out.println ("Snake " + getName () + " ate rabbit "
        + dinner.getName () + ".");
}

/**
 * Locates the specified snake and tells it to eat a rabbit.
 */
public static void eat (EntityManager em, String filter)
{
    em.getTransaction ().begin ();

    // Find the desired snake(s) in the data store.
    Query query = em.createQuery (filter);
    List<Snake> results = query.getResultList ();
    if (results.isEmpty ())
    {
        System.out.println ("No snakes matching '" + filter + "' found");
        return;
    }

    Query uneatenQuery = em.createQuery
        ("select r from Rabbit r where r.isDead = false");
    Random random = new Random ();
    for (Snake snake : results)
    {
        // Run a query for a rabbit whose 'isDead' field indicates
        // that it is alive.
        List<Rabbit> menu = uneatenQuery.getResultList ();
        if (menu.isEmpty ())
        {
            System.out.println ("No live rabbits in DB.");
            break;
        }

        // Select a random rabbit from the list.
        Rabbit dinner = menu.get (random.nextInt (menu.size ()));

        // Perform the eating.
        System.out.println (snake + " is eating:");
        snake.eat (dinner);
    }

    em.getTransaction ().commit ();
}

public static void main (String[] args)
{
    if (args.length == 2 && args[0].equals ("eat"))
    {
        EntityManagerFactory emf = Persistence.
            createEntityManagerFactory (null);
        EntityManager em = emf.createEntityManager ();
        eat (em, args[1]);
        em.close ();
        emf.close ();
        return;
    }

    // If we get here, something went wrong.
    System.out.println ("Usage:");
    System.out.println (" java tutorial.persistence.Snake eat "
        + "\"snakequery\"");
}

```

2. Add an eater field to Rabbit.java, and a getter and setter.

```

@ManyToOne @JoinColumn(name="EATER_ID")
private Snake eater;

```

```

...
public Snake getEater ()
{
    return eater;
}

public void setEater (Snake snake)
{
    eater = snake;
}

```

The `@ManyToOne` annotation indicates that this is the many side of the bidirectional relation. The many side must always be the owner in this type of relation. The `@JoinColumn` describes the foreign key that joins the rabbit table to the snake table. The rabbit table has an `EATER_ID` column that is a foreign key to the `ID` primary key column of the snake table.

3. Compile `Snake.java` and `Rabbit.java` and enhance the classes.

```

javac Snake.java Rabbit.java
kodoc -p persistence.xml Snake.java Rabbit.java

```

4. Refresh the mappings and database.

```

mappingtool -p persistence.xml Snake.java Rabbit.java

```

Now, experiment with the following commands:

```

java tutorial.persistence.Snake eat "select s from Snake s where s.name = '<name>'"
java tutorial.persistence.AnimalMaintenance details "select s from Snake s where s.name = '<name>'"

```

1.2.5.1. Complex Queries

Imagine that one of the snakes in the database was named Killer. To find out which rabbits Killer ate, we could run either of the following two queries:

```

java tutorial.persistence.AnimalMaintenance details "select s from Snake s where s.name = 'Killer'"
java tutorial.persistence.AnimalMaintenance list "select r from Rabbit r where r.eater.name = 'Killer'"

```

The first query is snake-centric - the query runs against the `Snake` class, looking for all snakes named Killer and providing a detailed listing of them. The second is rabbit-centric - it examines the rabbits in the database for instances whose `eater` is named Killer. This second query demonstrates that simple java 'dot' syntax is used when traversing an to-one field in a query.

It is also possible to traverse collection fields. Imagine that there was a rabbit called Roger in the datastore and that one of the snakes ate it. In order to determine who ate Roger Rabbit, you could run a query like this:

```
java tutorial.persistence.AnimalMaintenance details "select s from Snake s inner join s.giTract r where r.name = 'Roger'"
```

1.2.6. Extra Features

Congratulations! You are now the proud author of a pet store inventory suite. Now that you have all the major features of the pet store software implemented, it's time to add some extra features. You're on your own; think of some features that you think a pet store should have, or just explore the features of JPA.

Here are a couple of suggestions to get you started:

- Animal pricing.

Modify `Animal` to contain an inventory cost and a resale price. Calculate the real dollar amount eaten by the snakes (the sum of the inventory costs of all the consumed rabbits), and the cost assuming that all the eaten rabbits would have been sold had they been alive. Ignore the fact that the rabbits, had they lived, would have created more rabbits, and the implications of the reduced food costs due to the not-quite-as-hungry snakes and the smaller number of rabbits.

- Dog categorization.

Modify `Dog` to have a relation to a new class called `Breed`, which contains a name identifying the breed of the dog and a description of the breed. Put together an admin tool for breeds and for associating dogs and breeds.

1.3. J2EE Tutorial

By deploying Kodo into a J2EE environment, you can maintain the simplicity and performance of Kodo, while leveraging J2EE technologies such as container managed transactions (JTA/JTS), enterprise objects with remote invocation (EJB), and managed deployment of multi-tiered applications via an application server. This tutorial will demonstrate how to deploy Kodo-based J2EE applications and showcase some basic enterprise JPA design techniques. The tutorial's sample application models a basic garage catalog system. While the application is relatively trivial, the code has been constructed to illustrate simple patterns and solutions to common problems when using Kodo in an enterprise environment.

1.3.1. Prerequisites for the Kodo J2EE Tutorial

This tutorial assumes that you have installed Kodo and setup your classpath according to the installation instructions appropriate for your platform. In addition, this tutorial requires that you have installed and configured a J2EE-compliant application server, such as WebLogic or JBoss, running on JDK 1.5. If you use a different application server not listed here, this tutorial may be adaptable to your application server with small changes; refer to your application server's documentation for any specific classpath and deployment descriptor requirements.

This tutorial assumes a reasonable level of experience with Kodo and JPA. We provide a number of other tutorials for basic concepts, including enhancement, schema mapping, and configuration. This tutorial also assumes a basic level of experience with J2EE components, including session beans, JNDI, JSP, and EAR/WAR/JAR packaging. Sun and/or your application server company may provide tutorials to get familiar with these components.

In addition, this tutorial uses Ant to build the deployment archives. While this is the preferred way of building a deployment of the tutorial, one can easily build the appropriate JAR, WAR, and EAR files by hand, although that is outside the scope of this document.

1.3.2. J2EE Installation Types

Every application server has a different installation process for installing J2EE components. Kodo can be installed in a number of

ways, which may or may not be appropriate to your application server. While this document focuses mainly upon using Kodo as a JCA resource, there are other ways to use Kodo in a J2EE environment.

- JPA: J2EE 5 allows for the automatic injection of `EntityManager` instances into the J2EE context.
- JCA: Kodo implements the JCA 1.0 spec, and the `kodo-persistence.rar` file that comes in the `jca/persistence` directory of the distribution can be installed as any other JCA connection resource. This is the preferred way to integrate Kodo into a pre-J2EE 5 environment. It allows for simple installation (usually involving uploading or copying `kodo-persistence.rar` into the application server's deployment directory), and guided configuration on many appservers.
- Manual Binding into JNDI: Your application may require some needs in initializing Kodo that go beyond the JPA and JCA specifications. In this case, you can manually instantiate Kodo and place it into the JNDI tree. This process, however, is not seamless and can require a fair bit of custom application server code to bind an instance of `kodo.persistence.EntityManagerFactoryImpl` into JNDI.

1.3.3. Installing Kodo Into Pre-J2EE 5 Application Servers

Section 8.1.3, “Kodo JPA JCA Deployment” [527] goes over the steps to install Kodo on your application server of choice via the Java Connector Architecture (JCA). If your application server does not support JCA, see **Section 8.1.5, “Non-JCA Application Server Deployment”** [533] and **Section 8.1.1, “Standalone Deployment”** [527] for other installation options.

1.3.4. Installing the J2EE Sample Application

Installing the sample application involves first compiling and building a deployment archive (.ear) file. This file then needs to be deployed into your application server.

1.3.4.1. Compiling and Building The Sample Application

Navigate to the `samples/persistence/j2ee` directory of your Kodo installation.

Ensure that the JNDI name in the `setSessionContext()` method in `ejb/CarBean.java` matches your JCA installation. This defaults to `java:/kodo-ejb`, but the actual value will depend on the configuration of your JCA deploy and your application server's JNDI context. E.g. the default name for a WebLogic 9 install would be simply `kodo-ejb`.

Compile the source files in place both in this base directory as well as the nested `ejb` and `jsp` directories:

```
javac *.java  ejb/*.java  jsp/*.java
```

Enhance the `Car` class.

```
kodoc -p persistence.xml Car.java
```

Run the mapping tool; make sure that your `META-INF/persistence.xml` file includes the same connection information (e.g. `Driver`, `URL`, etc.) as your JCA installation. You should update your JCA configuration to include `samples.persistence.j2ee.Car` in the **MetaDataFactory** property:

```
<config-property name="MetaDataFactory">Types=samples.persistence.j2ee.Car</config-property/>
```

```
mappingtool -p persistence.xml Car.java
```

Build an J2EE application archive by running Ant against the `build.xml`. This will create `kodo-persistence-j2ee-sample.ear`. This ear can now be deployed to your appserver. Be sure to add the class `samples.j2ee.Car` to the `kodo.PersistentClasses` Kodo configuration property. This will automatically register the Entity.

```
ant -f build.xml
```

1.3.4.2. Deploying Sample To JBoss

Place the ear file in the `deploy` directory of your JBoss installation. You can use the above hints to view the JNDI tree to see if `samples.j2ee.ejb.CarHome` was deployed to JNDI.

1.3.4.3. Deploying Sample To WebLogic 9

Place the ear file in the `autodeploy` directory of your WebLogic domain. Production mode (see your `startWebLogic.sh/cmd` file) should be set to false to enable auto-deployment. If the application was installed correctly, you should see `kodo-persistence-j2ee-sample` listed in the Deployments section of the admin console. In addition you should find `CarHome` listed in the JNDI tree under `AdminServer->samples->j2ee->ejb`. Ensure that you have added the class `samples.j2ee.Car` to the `kodo.PersistentClasses` Kodo configuration property in the `META-INF/ra.xml` file.

1.3.5. Using The Sample Application

The sample application installs itself into the web layer at the context root of `sample`. By browsing to `http://yourserver:yourport/kodo-persistence-j2ee-sample`, you should be presented with a simple list page with no cars. You can edit, add, delete car instances. In addition, you can query on the underlying Car instances by passing in an JPQL query into the marked form (such as `select car from Car car where car.model="Some Model"`).

1.3.6. Sample Architecture

The garage application is a simple enterprise application that demonstrates some of the basic concepts necessary when using Kodo in the enterprise layer.

The core model wraps a stateless session bean facade around an entity. Using a session bean provides both a remote interface for various clients as well as providing a transactional context in which to work (and thus avoiding any explicit transactional code).

This session bean uses the JPA's detachement capabilities to provide an automatic Data Transfer Object mechanism for the primary communication between the application server and the (possibly remote) client. The `Car` instance will be used as the primary object upon which the client will work.

- `samples/persistence/j2ee/Car.java`: The core of the sample application. This is the entity class that Kodo will use to persist the application data. Instances of this class will also fill the role of data transfer object (DTO) for EJB clients. To accomplish this, `Car` implements `java.io.Serializable` so that remote clients can access cars as parameters and return values from the EJB.
- `samples/persistence/j2ee/jsp/SampleUtilities.java`: This is a simple facade to aggregate some common J2EE behavior into some static methods. By placing all of the functionality into a single facade class, we can reduce code maintenance of our JSPs.

- `samples/persistence/j2ee/ejb/Car*.java`: The source for the `CarEJB` session bean. Clients can use the `CarHome` and `CarRemote` interfaces to find, manipulate, and persist changes to `Car` transfer object instances. By using J2EE transactional features, the implementation code in `CarBean.java` can be focused almost entirely upon business and persistence logic without worrying about transactions.
- `samples/persistence/j2ee/jsp/*.jsp`: The web presentation client. These JSPs are not aware of the JPA; they simply use the `CarEJB` session bean and the `Car` transfer object to do all the work.
- `samples/persistence/j2ee/resources/*`: Files required to deploy to the various appservers, including J2EE deployment descriptors, WAR/EJB/EAR descriptors, as well as appserver specific files.
- `samples/persistence/j2ee/build.xml`: A simple Ant build file to help in creating a J2EE EAR file.

1.3.7. Code Notes and J2EE Tips

1. Entity classes are excellent candidates for the Data Transfer Object Pattern. This pattern attempts to reduce network load, as well as group business logic into concise units. For example, `CarBean.edit` allows you to ensure that all values are correct before committing a transaction, instead of sequentially calling getters and setters on the session facade. This is especially true when using RMI such as from a Swing based application connecting to an application server.

`CarEJB` works as a session bean facade to demarcate transactions, provide finder methods, and encapsulate complex business logic at the server level.

2. `EntityManager.close()` should be called at the end of every EJB method. In addition to ensuring that your code will not attempt to access a closed `EntityManager`, it allows Kodo to free up unused resources. Since an `EntityManager` is created for every transaction, this can increase the scalability of your application.
3. You should not use `EntityManager.getTransaction()` when using JTA to manage your transactions. Instead, Kodo will integrate with JTA to automatically govern transactions based on your EJB transaction configuration.
4. While serialization of entity instances is relatively straightforward, there are several things to keep in mind:
 - While "default fetch group" values will always be returned to the client upon serialization, lazily loaded fields will not as the `EntityManager` will have been closed before those fields attempt to serialize. You can either access those fields before serialization, configure the fetch type of the relationships, or configure your JPQL queries to eagerly fetch data.
5. It is not necessarily required that you use EJBs and container-managed transactions to demarcate transactions, although that is probably the most common method. In EJBs using bean managed transactions, you can control transactions through the `javax.transaction.UserTransaction` interface. Furthermore, outside of session beans you can control the JPA layer's transaction via the `javax.persistence.EntityTransaction` interface.
6. `EntityManagers` are allocated on a per-Transaction basis. Calling `getEntityManager` from the same `EntityManagerFactory` within the same EJB method call will always return the same entity manager, although the user-visible object may be proxied, so might not compare equal using the `==` operator.

Chapter 2. JDO Tutorials

2.1. Kodo JDO Tutorials

These tutorials provide step-by-step examples of how to use various facets of the Kodo JDO system. They assume a general knowledge of JDO and Java. For more information on these subjects, see the following URLs:

- [Sun's Java site](#)
- [JDO Overview Document](#)
- [Links to JDO Resources](#)

2.1.1. Tutorial Requirements

These tutorials require that JDK 1.2 or greater be installed on your computer, and that `java` and `javac` are in your `PATH` when you open a command shell. See [Chapter 2, *Kodo Installation* \[4\]](#) of Part I of this manual (the Kodo README) or see the [README.txt](#) included in the root directory of your download for more information on requirements and installation procedures.

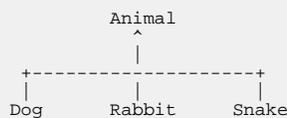
2.2. Kodo JDO Tutorial

In this tutorial you will become familiar with the basic tools and development processes under Kodo by creating a simple JDO application.

2.2.1. The Pet Shop

Imagine that you have decided to create a software toolkit to be used by pet shop operators. This toolkit must provide a number of solutions to common problems encountered at pet shops. Industry analysts indicate that the three most desired features are inventory maintenance, inventory growth simulation, and behavioral analysis. Not one to question the sage advice of experts, you choose to attack these three problems first.

According to the aforementioned experts, most pet shops focus on three types of animals only: dogs, rabbits, and snakes. This ontology suggests the following class hierarchy:



2.2.1.1. Included Files

We have provided an implementation of `Animal` and `Dog` classes, plus some helper classes and files to create the initial schema and populate the database with some sample dogs. Let's take a closer look at these classes.

- `tutorial.jdo.AnimalMaintenance` : Provides some utility methods for examining and manipulating the animals stored in the database. We will fill in method definitions in [Section 2.2.3, "Inventory Maintenance" \[349\]](#)

- `tutorial.jdo.Animal`: This is the superclass of all animals that this pet store software can handle.
- `tutorial.jdo.Dog`: Contains data and methods specific to dogs.
- `tutorial.jdo.Rabbit`: Contains data and methods specific to rabbits. It will be used in [Section 2.2.4, “Inventory Growth”](#) [352]
- `tutorial.jdo.Snake`: Contains data and methods specific to snakes. It will be used in [Section 2.2.5, “Behavioral Analysis”](#) [354]
- `package.jdo`: This is a JDO metadata file that defines which types should be enhanced into persistence-capable or persistence-aware classes. For more information on JDO metadata, consult [Chapter 5, Metadata](#) [193] of the JDO Overview.
- `../META-INF/jdo.properties`: This properties file contains Kodo-specific and standard JDO configuration settings.

Important

You must ensure you have valid BEA license key available to Kodo. For further details on Kodo licensing, see [Section 2.2, “License Configuration”](#) [393]

- `solutions`: The solutions directory contains the complete solutions to this tutorial, including finished versions of the `.java` files listed above and a correct `package.jdo` metadata file.

2.2.1.2. Important Utilities

- `java`: Runs main methods in specified Java classes.
- `javac`: Compiles `.java` files into `.class` files that can be executed by `java`.
- `jdoc`: Runs the Kodo JDO enhancer against the specified classes. More information is available in [Section 5.2, “Enhancement”](#) [452] of the Reference Guide.
- `mappingtool`: A utility that can be used to create and maintain the object-relational mappings and schema of all persistent classes in a JDBC-compliant datastore. This functionality allows the underlying mappings and schema to be easily kept up-to-date with the Java classes in the system. See [Chapter 7, Mapping](#) [489] of the Reference Guide for more information.

2.2.2. Getting Started

Let's compile the initial classes and see them in action. To do so, we must compile the `.java` files, as we would with any Java project, and then pass the resulting classes through the JDO enhancer:

Note

Be sure that your `CLASSPATH` is set correctly. Please see [Section 2.2.7, “Windows Installation”](#) [5] of Part I of this manual for Windows, or [Section 2.2.8, “POSIX \(Linux, Solaris, Mac OS X, Windows with cygwin, etc.\) Installation”](#) [5] of Part I of this manual for POSIX (Linux, Solaris, Cygwin, etc.). Detailed information on which libraries are needed can be found in [Appendix 7, Development and Runtime Libraries](#) [651]. Note, also, that your Kodo install directory should be in the `CLASSPATH`, as the tutorial classes are located in the `tutorial/jdo` directory under your Kodo install directory, and are in the `tutorial.jdo` package.

1. Change to the `tutorial/jdo` directory.

All examples throughout the tutorial assume that you are in this directory.

2. Examine `Animal.java`, `Dog.java`, and `SeedDatabase.java`

These files are good examples of the simplicity JDO engenders. As noted earlier, persisting an object or manipulating an object's persistent data requires almost no JDO-specific code. For a very simple example of creating persistent objects, please see the main method of `SeedDatabase.java`. Note the objects are created with normal Java constructors. The files `Animal.java` and `Dog.java` are also good examples of how JDO allows you to manipulate persistent data without writing any specific JDO code.

3. Compile the `.java` files.

```
javac *.java
```

You can use any java compiler instead of **javac**.

4. Enhance the JDO classes.

```
jdoc -p jdo.properties package.jdo
```

This step runs the Kodo enhancer on the `package.jdo` file mentioned above. The `package.jdo` file contains an enumeration of all the classes that should be JDO enhanced. The Kodo enhancer will examine the metadata defined in this file and enhance all classes listed in it appropriately. See [Section 5.2, “Enhancement” \[452\]](#) of the Reference Guide for more information on the JDO enhancer, including how to use Kodo's automatic runtime enhancement.

Note

The `-p` flag points the enhancer to your `jdo.properties` configuration file. All Kodo JDO tools look for default configuration in a resource called `kodo.properties` or `META-INF/kodo.properties`. Thus you can avoid passing the `-p` argument to tools by using this configuration file name in place of `jdo.properties`. See [Chapter 2, Configuration \[393\]](#) of the Reference Guide for details on Kodo configuration.

2.2.2.1. Configuring the Datastore

Now that we've compiled the source files and enhanced the JDO classes, we're ready to set up the database. **Hypersonic SQL**, a pure Java relational database, is included in the Kodo distribution. We have included this database because it is simple to set up and has a small memory footprint; however, you can use this tutorial with any of the relational databases that we support. You can also write your own plugin for any database that we do not support. For the sake of simplicity, this tutorial only describes how to set up connectivity to a Hypersonic SQL database. For more information on how to connect to a different database or how to add support for other databases, see [Chapter 4, JDBC \[423\]](#) of the Reference Guide.

1. Create the object-relational mappings and database schema.

```
mappingtool -p jdo.properties package.jdo
```

This command creates object-relational mappings for the classes listed in `package.jdo`, and at the same time propagates the necessary schema to the database configured in `jdo.properties`. If you are using the default Hypersonic SQL setup, the first time you run the mapping tool Hypersonic will create `tutorial_database.properties` and `tutorial_database.script` database files in your current directory. To delete the database, just delete these files.

By default, JDO stores object-relational mapping information in your `.jdo` files. As you will see in the **Reverse Mapping Tool Tutorial**, you can also configure Kodo to store object-relational mappings in separate files or in a database table. **Chapter 7, Mapping [489]** of the Reference Guide describes your mapping options in detail.

If you'd like to see the mapping information Kodo has just created, examine the `package.jdo` file. **Chapter 15, Mapping Metadata [260]** of the JDO Overview will help you understand mapping XML, should the need ever arise. Most Kodo development does not require any knowledge of mappings.

If you are curious, you can also view the schema Kodo created for the tutorial classes with Kodo's schema tool:

```
schematool -p jdo.properties -a reflect -f tmp.schema
```

This will create a `tmp.schema` file with an XML representation of the database schema. The XML should be self explanatory; see **Section 4.14, "XML Schema Format" [447]** of the Reference Guide for details. You may delete the `tmp.schema` file before proceeding.

2. Populate the database with sample data.

```
java tutorial.jdo.SeedDatabase
```

Congratulations! You have now created a JDO-accessible persistent store, and seeded it with some sample data.

2.2.3. Inventory Maintenance

The most important element of a successful pet store product, say the experts, is an inventory maintenance mechanism. So, let's work on the `Animal` and `Dog` classes a bit to permit user interaction with the database.

This chapter should familiarize you with some of the basics of the **JDO specification** and the mechanics of compiling and enhancing persistence-capable objects. You will also become familiar with the mapping tool for propagating the JDO schema into the database.

First, let's add some code to `AnimalMaintenance.java` that allows us to examine the animals currently in the database.

1. Add code to `AnimalMaintenance.java`.

Modify the `getAnimals` method of `AnimalMaintenance.java` to look like this:

```
/**
 * Return a list of animals that match the specified query filter.
 *
 * @param filter      the JDO filter to apply to the query
 * @param cls         the class of animal to query on
 * @param pm          the PersistenceManager to obtain the query from
 */
public static List getAnimals (String filter, Class cls,
                             PersistenceManager pm)
{
    // Execute a query for the specified class and filter.
```

```
        Query query = pm.newQuery (cls, filter);
        return (List) query.execute ();
    }
```

2. Compile `AnimalMaintenance.java`.

```
javac AnimalMaintenance.java
```

3. Take a look at the animals in the database.

```
java tutorial.jdo.AnimalMaintenance list Animal
```

Notice that `list` optionally takes a query filter. Let's explore the database some more, this time using filters:

```
java tutorial.jdo.AnimalMaintenance list Animal "name == 'Binney'"
java tutorial.jdo.AnimalMaintenance list Animal "price <= 50"
```

The JDO query language is designed to look and behave much like boolean expressions in Java. The name and price fields identified in the above queries map to the member fields of those names in `tutorial.jdo.Animal`. More details on JDO query syntax is available in **Chapter 11, Query [232]** of the JDO Overview.

Great! Now that we can see the contents of the database, let's add some code that lets us add and remove animals.

2.2.3.1. Persisting Objects

As new dogs are born or acquired, the store owner will need to add new records to the inventory database. In this section, we'll write the code to handle additions through the `tutorial.jdo.AnimalMaintenance` class.

This section will familiarize you with the mechanism for storing persistence-capable objects in a JDO persistence manager. We will create a new dog, obtain a `Transaction` from a `PersistenceManager`, and, within the transaction, make the new dog object persistent.

`tutorial.jdo.AnimalMaintenance` provides a reflection-based facility for creating any type of animal, provided that the animal has a two-argument constructor whose first argument corresponds to the name of the animal and whose second argument is an implementation-specific primitive. This reflection-based system is in place to keep this tutorial short and remove repetitive creation mechanisms. It is not a required part of the JDO specification.

1. Add the following code to `AnimalMaintenance.java`.

Modify the `persistObject` method of `AnimalMaintenance.java` to look like this:

```
/**
 * Performs the actual JDO work of putting <code>object</code>
 * into the datastore.
 *
 * @param object the object to persist in the datastore
```

```

    */
    public static void persistObject (Object object)
    {
        // Get a PersistenceManagerFactory and PersistenceManager.
        PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory
            ("META-INF/jdo.properties");
        PersistenceManager pm = pmf.getPersistenceManager ();

        // Obtain a transaction and mark the beginning
        // of the unit of work boundary.
        Transaction transaction = pm.currentTransaction ();
        transaction.begin ();

        pm.makePersistent (object);

        // Mark the end of the unit of work boundary,
        // and record all inserts in the database.
        transaction.commit ();

        System.out.println ("Added " + object);

        // Close the PersistenceManager and PersistenceManagerFactory.
        pm.close ();
        pmf.close ();
    }

```

2. Recompile AnimalMaintenance.java.

```
javac AnimalMaintenance.java
```

You now have a mechanism for adding new dogs to the database. Go ahead and add some by running **java tutorial.jdo.AnimalMaintenance add Dog <name> <price>** For example:

```
java tutorial.jdo.AnimalMaintenance add Dog Fluffy 35
```

You can view the contents of the database with:

```
java tutorial.jdo.AnimalMaintenance list Dog
```

2.2.3.2. Deleting Objects

What if someone decides to buy one of the dogs? The store owner will need to remove that animal from the database, since it is no longer in the inventory.

This section demonstrates how to remove data from the datastore.

1. Add the following code to AnimalMaintenance.java .

Modify the `deleteObjects` method of `AnimalMaintenance.java` to look like this:

```

/**
 * Performs the actual JDO work of removing
 * <code>objects</code> from the datastore.

```

```
*
* @param objects    the objects to persist in the datastore
* @param pm         the PersistenceManager to delete with
*/
public static void deleteObjects (Collection objects, PersistenceManager pm)
{
    // Obtain a transaction and mark the beginning of the
    // unit of work boundary.
    Transaction transaction = pm.currentTransaction ();
    transaction.begin ();

    for (Iterator iter = objects.iterator (); iter.hasNext (); )
        System.out.println ("Removed animal: " + iter.next ());

    // This method removes the objects in 'objects' from the datastore.
    pm.deletePersistentAll (objects);

    // Mark the end of the unit of work boundary, and record all
    // deletes in the database.
    transaction.commit ();
}
```

2. Recompile `AnimalMaintenance.java`.

```
javac AnimalMaintenance.java
```

3. Remove some animals from the database.

```
java tutorial.jdo.AnimalMaintenance remove Animal <query>
```

Where *<query>* is a query string like those used for listing animals above.

All right. We now have a basic pet shop inventory management system. From this base, we will add some of the more advanced features suggested by our industry experts.

2.2.4. Inventory Growth

Now that we have the basic pet store framework in place, let's add support for the next pet in our list: the rabbit. The rabbit is a bit different than the dog; pet stores sell them all for the same price, but gender is critically important since rabbits reproduce rather easily and quickly. Let's put together a class representing a rabbit.

In this chapter, you will see some more queries and write a bidirectional relation between objects.

Provided with this tutorial is a file called `Rabbit.java` which contains a sample `Rabbit` implementation. Let's get it compiled and loaded:

1. Examine and compile `Rabbit.java`.

```
javac Rabbit.java
```

2. Add an entry for Rabbit to `package.jdo`.

The Rabbit class above contains a bidirectional relationship between parents and children. From the Java side of things, a bidirectional relationship is simply a pair of fields that are conceptually linked. There is no special Java work necessary to express bidirectionality. However, you must identify the relationship as bidirectional in the **JDO metadata** for the mapping tool to create the most efficient schema. Insert the snippet below into the `package.jdo` file. It identifies both the type of data in the collection (the `element-type` attribute) and the name of the other side of the relation. The `mapped-by` attribute for identifying bidirectional relations is part of JDOR, a subset of the JDO specification for relational databases. For more information on JDOR, consult **Chapter 14, JDOR [259]** of the JDO Overview.

Add the following code immediately *before* the "`</package>`" line in the `package.jdo` file.

```
<class name="Rabbit">
  <field name="parents">
    <collection element-type="Rabbit"/>
  </field>
  <field name="children" mapped-by="parents">
    <collection element-type="Rabbit"/>
  </field>
</class>
```

3. Enhance the Rabbit class.

```
jdoc -p jdo.properties Rabbit.java
```

4. Refresh the object-relational mappings and database schema.

```
mappingtool -p jdo.properties Rabbit.java
```

Now that we have a Rabbit class, let's get some preliminary rabbit data into the database.

1. Create some rabbits.

Run the following commands a few times to add some male and female rabbits to the database:

```
java tutorial.jdo.AnimalMaintenance add Rabbit <name> false
java tutorial.jdo.AnimalMaintenance add Rabbit <name> true
```

Now run some breeding iterations.

```
java tutorial.jdo.Rabbit breed 2
```

2. Look at your new rabbits.

```
java tutorial.jdo.AnimalMaintenance list Rabbit  
java tutorial.jdo.AnimalMaintenance details Rabbit ""
```

2.2.5. Behavioral Analysis

Often, pet stores sell snakes as well as rabbits and dogs. Pet stores are primarily concerned with a snake's length; much like rabbits, pet store operators usually sell them all for a flat rate.

This chapter demonstrates more queries, schema manipulation, and additional relation types.

Provided with this tutorial is a file called `Snake.java` which contains a sample `Snake` implementation. Let's get it compiled and loaded:

1. Examine and compile `Snake.java`.

```
javac Snake.java
```

2. Add `tutorial.jdo.Snake` to `package.jdo`.

```
<class name="Snake"/>
```

3. Enhance the class.

```
jdoc -p jdo.properties Snake.java
```

4. Refresh the mappings and database.

As we have created a new persistence-capable class, we must map it to the database and change the schema to match. So run the mapping tool:

```
mappingtool -p jdo.properties Snake.java
```

Once you have compiled everything, add a few snakes to the database using:

```
java tutorial.jdo.AnimalMaintenance add Snake <name> <length>
```

Where *<name>* is the name and *<length>* is the length in feet for the new snake. To see the new snakes in the database, run:

```
java tutorial.jdo.AnimalMaintenance list Snake
```

Unfortunately for the massively developing rabbit population, snakes often eat rabbits. Any good inventory system should be able to capture this behavior. So, let's add some code to `Snake.java` to support the snake's eating behavior.

First, let's modify `Snake.java` to contain a list of eaten rabbits.

1. Add the following code snippet to `Snake.java`.

```
// *** Add this member variable declaration. ***
private Set giTract = new HashSet ();

...

// *** Modify toString (boolean) to output the giTract list. ***
public String toString (boolean detailed)
{
    StringBuffer buf = new StringBuffer (1024);
    buf.append ("Snake ").append (getName ());

    if (detailed)
    {
        buf.append (" (").append (length).append (" feet long) sells for ");
        buf.append (getPrice ()).append (" dollars.");
        buf.append (" Its gastrointestinal tract contains:\n");
        for (Iterator iter = giTract.iterator (); iter.hasNext ());
            buf.append ("\t").append (iter.next ()).append ("\n");
    }
    else
        buf.append ("; ate " + giTract.size () + " rabbits.");

    return buf.toString ();
}

...

// *** Add these methods. ***
/**
 * Kills the specified rabbit and eats it.
 */
public void eat (Rabbit dinner)
{
    // Consume the rabbit.
    dinner.kill ();
    dinner.eater = this;
    giTract.add (dinner);
    System.out.println ("Snake " + getName () + " ate rabbit "
        + dinner.getName () + ".");
}

/**
 * Locates the specified snake and tells it to eat a rabbit.
 */
public static void eat (String filter)
{
    PersistenceManagerFactory pmf = JDOHelper.
        getPersistenceManagerFactory ("META-INF/jdo.properties");
    PersistenceManager pm = pmf.getPersistenceManager ();
    Transaction transaction = pm.currentTransaction ();
    transaction.begin ();

    // Find the desired snake(s) in the datastore.
    Query query = pm.newQuery (Snake.class, filter);
    List results = (List) query.execute ();
    if (results.isEmpty ())
    {
        System.out.println ("No snakes matching '" + filter + "' found");
        return;
    }

    Query uneatenQuery = pm.newQuery (Rabbit.class, "isDead == false");
    Random random = new Random ();
    for (Iterator iter = results.iterator (); iter.hasNext ());
    {
        // Run a query for a rabbit whose 'isDead' field indicates
        // that it is alive.
        List menu = (List) uneatenQuery.execute ();
    }
}
}
```

```

        if (menu.isEmpty ())
        {
            System.out.println ("No live rabbits in DB.");
            break;
        }

        // Select a random rabbit from the list.
        Rabbit dinner = (Rabbit) menu.get (random.nextInt (menu.size ()));

        // Perform the eating.
        Snake snake = (Snake) iter.next ();
        System.out.println (snake + " is eating:");
        snake.eat (dinner);
    }

    transaction.commit ();
    pm.close ();
    pmf.close ();
}

public static void main (String [] args)
{
    if (args.length == 2 && args[0].equals ("eat"))
    {
        eat (args[1]);
        return;
    }

    // If we get here, something went wrong.
    System.out.println ("Usage:");
    System.out.println (" java tutorial.jdo.Snake eat 'snakequery'");
}

```

2. Add an eater field to Rabbit.java.

Notice that we are making this field `protected`, and that we set it from the Snake class. This demonstrates that it is possible to directly access public or protected persistent fields in other classes. This is not recommended practice, however, because it means that all classes that access this field directly must be JDO enhanced, even if they are not persistence-capable.

Add the following member variable to Rabbit.java :

```
protected Snake eater;
```

3. Add metadata to package.jdo.

Notice the `giTract` declaration in Snake.java: it is a simple Java collection declaration. As with the `parents` and `children` sets in Rabbit.java, we augment this declaration with JDO metadata.

```

<class name="Snake">
    ...
    <field name="giTract" mapped-by="eater">
        <collection element-type="Rabbit"/>
    </field>
</class>

```

Note that we specified a `mapped-by` attribute in this example. This is because the relation is bidirectional; that is, the rabbit has knowledge of which snake ate it. We could have left out the `eater` field and instead created a standard unidirectional relation. The metadata might have looked like this:

```

<class name="Snake">
    <field name="giTract">
        <collection element-type="Rabbit"/>
    </field>
</class>

```

```
</field>
</class>
```

For more information on types of relations, see [Section 15.11, “Field Mapping” \[287\]](#) of the JDO Overview.

4. Compile `Snake.java` and `Rabbit.java` and enhance the classes.

```
javac Snake.java Rabbit.java
jdoc -p jdo.properties Snake.java Rabbit.java
```

5. Refresh the mappings and database.

```
mappingtool -p jdo.properties Snake.java Rabbit.java
```

Now, experiment with the following commands:

```
java tutorial.jdo.Snake eat ""
java tutorial.jdo.AnimalMaintenance details Snake ""
```

2.2.5.1. Complex Queries

Imagine that one of the snakes in the database was named Killer. To find out which rabbits Killer ate, we could run either of the following two queries:

```
java tutorial.jdo.AnimalMaintenance details Snake "name == 'Killer'"
java tutorial.jdo.AnimalMaintenance list Rabbit "eater.name == 'Killer'"
```

The first query is snake-centric - the query runs against the `Snake` class, looking for all snakes named Killer and providing a detailed listing of them. The second is rabbit-centric - it examines the rabbits in the database for instances whose `eater` is named Killer. This second query demonstrates that simple Java 'dot' syntax is used when traversing a relation field in a query.

It is also possible to traverse collection fields. Imagine that there was a rabbit called Roger in the datastore and that one of the snakes ate it. In order to determine who ate Roger Rabbit, you could run a query like this:

```
java tutorial.jdo.AnimalMaintenance details Snake "giTract.contains (rabbit) && rabbit.name == 'Roger'"
```

Note the use of the `rabbit` variable above. Variables are tokens that represent any element of the collection that contains them. So in our query, the `rabbit` variable stands for any rabbit in the snake's `giTract` collection. We could have called the variable anything; just as in Java, JDO query variables do not have to follow a set naming pattern.

2.2.6. Extra Features

Congratulations! You are now the proud author of a pet store inventory suite. Now that you have all the major features of the pet store software implemented, it's time to add some extra features. You're on your own; think of some features that you think a pet store should have, or just explore the features of JDO.

Here are a couple of suggestions to get you started:

- Animal pricing.

Modify `Animal` to contain an inventory cost and a resale price. Calculate the real dollar amount eaten by the snakes (the sum of the inventory costs of all the consumed rabbits), and the cost assuming that all the eaten rabbits would have been sold had they been alive. Ignore the fact that the rabbits, had they lived, would have created more rabbits, and the implications of the reduced food costs due to the not-quite-as-hungry snakes and the smaller number of rabbits.

- Dog categorization.

Modify `Dog` to have a relation to a new class called `Breed`, which contains a name identifying the breed of the dog and a description of the breed. Put together an admin tool for breeds and for associating dogs and breeds.

2.3. Reverse Mapping Tool Tutorial

In this tutorial you will learn how to use Kodo's reverse mapping tool to reverse-engineer persistent classes from a database schema.

2.3.1. Magazine Shop

You run a shop that sells magazines. You store information about your inventory in a relational database with the following schema:

```
-- Holds information on available magazines
CREATE TABLE MAGAZINE (
  ISBN VARCHAR(8) NOT NULL,
  ISSUE INTEGER NOT NULL,
  NAME VARCHAR(255),
  PUBLISHER_NAME VARCHAR(255)
  PRICE FLOAT NOT NULL,
  PRIMARY KEY (ISBN, ISSUE)
  FOREIGN KEY (PUBLISHER_NAME) REFERENCES PUBLISHER (NAME)
);

-- Holds information on magazine articles
CREATE TABLE ARTICLE (
  TITLE VARCHAR(255) NOT NULL,
  AUTHOR_NAME VARCHAR(128),
  PRIMARY KEY (TITLE)
);

-- Holds all the subtitles of an article
CREATE TABLE ARTICLE_SUBTITLES (
  ARTICLE_TITLE VARCHAR(255),
  SUBTITLE VARCHAR(128),
  FOREIGN KEY (ARTICLE_TITLE) REFERENCES ARTICLE (TITLE)
);

-- Join table linking magazines and articles
CREATE TABLE MAGAZINE_ARTICLES (
  MAGAZINE_ISBN VARCHAR(8),
  MAGAZINE_ISSUE INTEGER,
  ARTICLE_TITLE VARCHAR(255),
  FOREIGN KEY (MAGAZINE_ISBN, MAGAZINE_ISSUE) REFERENCES MAGAZINE (ISBN, ISSUE),
  FOREIGN KEY (ARTICLE_TITLE) REFERENCES ARTICLE (TITLE)
);

-- Each magazine has a 1-1 relation to its publisher
CREATE TABLE PUBLISHER (
  NAME VARCHAR(255) NOT NULL,
  REVENUE FLOAT NOT NULL,
  PRIMARY KEY (NAME)
);
```

You've decided to write an application that will let you query your database through JDO.

2.3.2. Setup

If you haven't already, follow the instructions in **README.txt**. This will ensure that your `CLASSPATH` and other environmental variables are set up correctly. Once you've completed the installation instructions, change into the `reversetutorial/jdo` directory.

2.3.2.1. Tutorial Files

The tutorial uses the following files:

- `reversetutorial_database.properties` and `reversetutorial_database.script`: These files make up a Hypersonic file-based database with the schema outlined above. The database is already populated with lots of magazine data representing your shop's inventory.
- `reversetutorial.jdo.Finder`: Uses JDO to execute user-supplied query strings and output the matching persistent objects. This class relies on persistent classes that we haven't generated yet, so it won't compile immediately.
- `../META-INF/jdo.properties`: Properties file containing Kodo-specific and standard JDO configuration settings.

For this tutorial, make sure the following properties are set to the values below:

```
javax.jdo.option.ConnectionURL: jdbc:hsqldb:reversetutorial_database
javax.jdo.option.ConnectionDriverName: org.hsqldb.jdbcDriver
javax.jdo.option.ConnectionUserName: sa
javax.jdo.option.ConnectionPassword:
```

Important

You must ensure you have valid BEA license key available to Kodo. For further details on Kodo licensing, see [Section 2.2, "License Configuration" \[393\]](#)

- `solutions`: Contains the complete solutions to this tutorial, including all generated code.

2.3.2.2. Important Utilities

- `java`: Runs main methods in specified Java classes.
- `javac`: Compiles `.java` files into `.class` files that can be executed by `java`.
- `jdoc`: Runs the Kodo JDO enhancer against the specified classes. More information is available in [Section 5.2, "Enhancement" \[452\]](#) of the Reference Guide.

2.3.3. Generating Persistent Classes

Now it's time to turn your magazine database into persistent JDO classes mapped to each existing table. To accomplish this, we'll use Kodo's reverse mapping tools.

1. First, make sure that you are in the `reversetutorial/jdo` directory and that you've made the appropriate modifications to your `jdo.properties` file, as described in the previous section.
2. Now that we have our environment set up correctly, we're going to dump our existing schema to an XML document. This step is not strictly necessary for Hypersonic SQL, which provides good database metadata. Some databases, however, have faulty JDBC drivers, and Kodo is unable to gather enough information about the existing schema to create a good object model from it. In these cases, it is useful to dump the schema to XML, then modify the XML by hand to correct any errors introduced by the JDBC driver. If your schema doesn't use foreign key constraints, you may also want to add logical foreign keys to the XML file so that Kodo can create the corresponding relations between the persistent classes it generates.

To perform the schema-to-XML conversion, we're going to use the schema tool, which can be invoked via the included `schematool` shell script. The `-p` option points the tool at your `jdo.properties` configuration file. `-a` denotes the tool's action. Finally, the `-f` flag tells the tool what to name the XML file it creates:

```
schematool -p jdo.properties -a reflect -f schema.xml
```

The schema tool is described in detail in [Section 4.13, “Schema Tool” \[444\]](#) of the Reference Guide.

Note

All Kodo JDO tools look for default configuration in a resource called `kodo.properties` or `META-INF/kodo.properties`. Thus you can avoid passing the `-p` argument to tools by using this configuration file name in place of `jdo.properties`. See [Chapter 2, Configuration \[393\]](#) of the Reference Guide for details on Kodo configuration.

3. Examine the `schema.xml` XML file created by the schema tool. As you can see, it contains a complete representation of the schema for your magazine database. For the curious, this XML format is documented in [Section 4.14, “XML Schema Format” \[447\]](#) of the Reference Guide.
4. Typically, JDO stores persistence and mapping metadata together in the same file. As shop owner, you've decided that you want to keep your mapping metadata separate from your persistence metadata. JDO supports separating object-relational mapping from persistence metadata through the `javax.jdo.option.Mapping` property. This property sets a logical name for your mappings. When it is set, JDO looks for mapping metadata in special mapping files suffixed with this name. Using the `javax.jdo.option.Mapping` property, you can separate concerns and even define multiple mappings for the same classes. You have chosen `hsq1` as the logical name for your mappings to the supplied HSQL database. Add the following line to the `../META-INF/jdo.properties` file:

```
javax.jdo.option.Mapping: hsq1
```

5. Run the reverse mapping tool on the schema file. (If you do not supply the schema file to reverse map, the tool will run directly against the schema in the database). The tool can be run via the included `reversemappingtool` script. Use the `-pkg` flag to control the package name of the generated classes.

```
reversemappingtool -p jdo.properties -pkg reversetutorial.jdo schema.xml
```

The reverse mapping tool has many options and customization hooks. For a complete treatment of the tool, see [Section 7.2, “Reverse Mapping” \[495\]](#) of the Reference Guide.

Running the reverse mapping tool will generate `.java` files for each generated class, `.java` files for corresponding JDO application identity classes, a `package.jdo` file containing persistence metadata, and a `package-hsqldb.orm` file containing object-relational mapping metadata. [Chapter 5, Metadata \[193\]](#) and [Chapter 15, Mapping Metadata \[260\]](#) of the JDO Overview discuss JDO persistence and mapping metadata, respectively.

6. Examine the generated persistent classes. Notice that the reverse mapping tool has used column and foreign key data to create the appropriate persistent fields and relations between classes. Notice, too, that due to the transparency of JDO, the generated code is vanilla Java, with no trace of JDO-specific functionality.

Also examine `MagazineId`, the generated application identity class for `Magazine`. Note that it satisfies all of the requirements for application identity classes mandated by the JDO specification, including the `equals` and `hashCode` contracts. The other generated classes use JDO's single field identity, a special case of application identity for classes with only one primary key field. [Section 4.5, “JDO Identity” \[186\]](#) explains JDO identity types.

Finally, examine the `package.jdo` metadata file and `package-hsqldb.orm` mapping file. The former contains the necessary standard persistence metadata, while the latter maps the generated classes and their fields to the existing schema.

7. Compile the generated classes:

```
javac *.java
```

The reverse mapping tool has now created a complete JDO object model for your magazine shop's existing relational model. From now on, you can treat the generated classes just like any other JDO class. And that means you have to complete one additional step before you can use the classes for persistence: enhancement.

```
jdoc -p jdo.properties package.jdo
```

This step runs the Kodo JDO enhancer on the `package.jdo` file mentioned above. The Kodo enhancer will examine the file's metadata and enhance all listed classes appropriately. See [Section 5.2, “Enhancement” \[452\]](#) of the Reference Guide for more information on the enhancer, including how to use Kodo's automatic runtime enhancement.

Congratulations! You are now ready to use JDO to access your magazine data.

2.3.4. Using the Finder

The `reversetutorial.jdo.Finder` class lets you run queries in JDOQL (JDO's Java-centric query syntax) against the existing database:

```
java reversetutorial.jdo.Finder <jdoql-query>
```

JDOQL is discussed in [Chapter 11, Query \[232\]](#) of the JDO Overview. JDOQL looks exactly like Java boolean expressions. To find magazines matching a set of criteria in JDOQL, just specify conditions on the `reversetutorial.jdo.Magazine` class' persistent fields. Some examples of valid JDOQL queries for magazines include:

```
java revesetutorial.jdo.Finder "true" // use this to list all the magazines
java revesetutorial.jdo.Finder "price < 5.0"
java revesetutorial.jdo.Finder "name == 'Vogue' || issue > 1000"
java revesetutorial.jdo.Finder "name.startsWith ('V')"
```

To traverse object relations, just use Java's dot syntax:

```
java revesetutorial.jdo.Finder "publisher.name == 'Adventure' || publisher.revenue > 1000000"
```

To traverse collection relations, you have to use JDOQL variables. Variables are just placeholders for any member of a collection. For example, in the following query, `art` is a variable:

```
java revesetutorial.jdo.Finder "articles.contains (art) && art.title.startsWith ('JDO')"
```

The above query is equivalent to "find all magazines that have an article whose title starts with 'JDO'". For example, to find all magazines whose publisher published an article about "Surpassing Hubble" in any of its magazines:

```
java revesetutorial.jdo.Finder "publisher.magazines.contains (mag) && mag.articles.contains (art) && art.articleSubtitles.contains ('Surpassing Hubble')"
```

Have fun experimenting with additional queries.

Note

Remember to erase the `javax.jdo.option.Mapping` setting in `../META-INF/jdo.properties` before continuing with your own JDO experimentation, unless you'd like to continue to store your mapping information in separate mapping files rather than your `.jdo` metadata files.

2.4. J2EE Tutorial

By deploying Kodo into a J2EE environment, you can maintain the simplicity and performance of Kodo, while leveraging J2EE technologies such as container managed transactions (JTA/JTS), enterprise objects with remote invocation (EJB), and managed deployment of multi-tiered applications via an application server. This tutorial will demonstrate how to deploy Kodo-based J2EE applications and showcase some basic enterprise JDO design techniques. The tutorial's sample application attempts to model a basic garage catalog system. While the application is relatively trivial, the code has been constructed to illustrate simple patterns and solutions to common problems when using Kodo in an enterprise environment.

2.4.1. Prerequisites for the Kodo J2EE Tutorial

This tutorial assumes that you have installed Kodo and setup your classpath according to the installation instructions appropriate for your platform. In addition, this tutorial requires that you have installed and configured a J2EE-compliant application server, such as JBoss, WebSphere, WebLogic, Borland Enterprise Server, JRun, or SunONE / Sun JES. If you use a different application server, this tutorial may be adaptable to your application server with small changes; refer to your application server's documentation for any specific classpath and deployment descriptor requirements.

This tutorial assumes a reasonable level of experience with Kodo and JDO. We provide a number of other tutorials for basic Kodo and JDO concepts, including enhancement, schema mapping, and configuration. This tutorial also assumes a basic level of experience with J2EE components, including EJB, JNDI, JSP, and EAR/WAR/JAR packaging. Sun and/or your application server company may provide tutorials to get familiar with these components.

In addition, this tutorial uses Ant to build the deployment archives. While this is the preferred way of building a deployment of the tutorial, one can easily build the appropriate JAR, WAR, and EAR files by hand, although that is outside the scope of this document.

2.4.2. J2EE Installation Types

Every application server has a different installation process for installing J2EE components. Kodo can be installed in a number of ways, each of which may or may not be appropriate to your application server. While this document focuses mainly upon using Kodo as a JCA resource, there are other ways to use Kodo in a J2EE environment.

- **JCA:** Kodo implements the JCA 1.0 spec, and the `kodo.rar` file that comes in the `jca` directory of the distribution can be installed as any other JCA connection resource. This is the preferred way to integrate Kodo into a J2EE environment. It allows for simple installation (usually involving uploading or copying `kodo.rar` into the application server), guided configuration on many appservers, as well as dynamic reloading for upgrading Kodo to a newer version. The drawback is that older application servers have flaky or non-existent JCA support as the spec is relatively new.
- **PersistenceManagerFactoryImpl:** This remains the most compatible way to integrate Kodo into a J2EE environment, though this is not seamless and can require a fair bit of custom application server code to manually bind an instance of `PersistenceManagerFactoryImpl` into the JNDI tree. This is somewhat offset by avoiding JCA configuration issues as well as being able to tailor the binding and start-up process.

2.4.3. Installing Kodo

Section 8.1.4, “Kodo JDO JCA Deployment” [528] goes over the steps to install Kodo on your application server of choice via the Java Connector Architecture (JCA). If your application server does not support JCA, see **Section 8.1.5, “Non-JCA Application Server Deployment”** [533] and **Section 8.1.1, “Standalone Deployment”** [527] for other installation options.

2.4.4. Installing the J2EE Sample Application

Installing the sample application involves first compiling and building a deployment archive (`.ear`) file. This file then needs to be deployed into your application server.

2.4.4.1. Compiling and Building The Sample Application

Navigate to the `samples/jdo/j2ee` directory of your Kodo installation. Compile the source files in place both in this base directory as well as the nested `ejb` directory:

```
javac *.java  ejb/*.java
```

Enhance the Car class.

```
jdoc -p jdo.properties package.jdo
```

Run the mapping tool; make sure that your `META-INF/jdo.properties` file includes the same connection information (e.g. Driver, URL, etc.) as your installation:

```
mappingtool -p jdo.properties package.jdo
```

Configure options in `samples.properties` to match your JCA installation, most notably the JNDI name to which you have bound Kodo (it defaults to kodo).

Warning

This step (editing `samples.properties`) is *very important* as this value can be quite different for each appserver and each configuration.

Be sure that the setting you put for `pmf.jndi` matches not only your configured setting but also what your application server may prefix the configured name with.

JBoss, for example, will prefix the JNDI name with `java:/` and Borland Enterprise Server looks at the `serial://` context. Refer to your JNDI tree and the documentation for your application server for further details.

Build an J2EE application archive by running Ant against the `build.xml`. This will create `samplej2ee.ear`. This ear can now be deployed to your appserver.

```
ant -f build.xml
```

2.4.4.2. Deploying Sample To JBoss

Place the ear file in the `deploy` directory of your JBoss installation. You can use the above hints to view the JNDI tree to see if `samples.jdo.j2ee.ejb.CarHome` was deployed to JNDI.

2.4.4.3. Deploying Sample To WebLogic 6.1 to 7.x

Place the ear file in the `applications` directory of your WebLogic domain. Production mode (see your `startWebLogic.sh/cmd` file) should be set to false to enable auto-deployment. If the application was installed correctly, you should see `sample-ejb` listed in the Deployments/EJB section of the admin console. In addition you should find `CarHome` listed in the JNDI tree under `myserver->samples->j2ee->ejb`.

2.4.4.4. Deploying Sample To WebLogic 8.1

Create a new directory named `samplej2ee.ear` in the `applications` directory of your WebLogic domain. Extract the EAR file (without copying the EAR file) to this new directory:

```
applications> mkdir samplej2ee.ear
applications> cd samplej2ee.ear
samplej2ee.ear> jar -xvf /path/to/samplej2ee.ear
```

Deploy the application by using the admin console (Deployments -> Applications -> Deploy a new Application. Select `samplej2ee.ear` and deploy to the proper server targets. If you have installed the application correctly, you should find `Car-`

Home listed in the JNDI tree under `myserver->samples->j2ee->ejb` .

2.4.4.5. Deploying Sample To SunONE / Sun JES

Browse to the admin console in your web browser. Select `Applications / Enterprise Applications` from the left navigation panel. Select `Deploy...` and in the following screen upload the `samplej2ee.ear` file to the server. Apply your changes by selecting the link in the upper right portion of the page. You should now see `samplej2ee` listed in the `Enterprise Applications` folder of the navigation panel.

2.4.4.6. Deploying Sample To JRun

Browse to the admin console in your web browser. Select `J2EE Components` from the left navigation panel. Select `Add` under the `Enterprise Applications` heading. Select the `samplej2ee.ear` file and hit `Deploy`. You should now see `Sample-KodoJ2EE` listed in the top-level folder of the navigation panel. Select it, and then select the `sample-ejb.jar#CarEJB` component under `Enterprise JavaBeans` section, then change the `JNDI Name` for the bean from the default value to `samples.jdo.j2ee.ejb.CarHome` and hit `Apply`.

If the Kodo resource adapter and the sample EAR are both configured correctly, you should now be able to access your sample application at JRun's deploy URL (e.g., `http://localhost:8100/sample/`).

2.4.4.7. Deploying Sample To WebSphere

Browse to the admin console in your web browser. Select `Applications / Install New Application` from the left navigation panel. Select the path to your `samplej2ee.ear` file and press `Next`.

On the following screen, leave the options at the default and select `Next`. On the following screen (`Install New Application->Step 1`), ensure that the `Deploy EJBs` option is checked. Leave other options at their defaults.

Move on to `Step 2`. On this screen enter `samples.jdo.j2ee.ejb.CarHome` as the `JNDI name` for the `Car EJB`. Continue through the remaining steps leaving options at the defaults. Select `Finish` and ensure that the application is deployed correctly.

Save the changes to the domain configuration by either selecting the `Save` link that appears after the installation is complete or by selecting `Save` from the top menu.

To verify your installation, select `Applications / Enterprise Applications` from the left navigation panel. `Sample-KodoJ2EE` should be listed in the list. If the application has not started already, select the checkbox next to `Sample-KodoJ2EE` and select `Start`.

2.4.4.8. Deploying Sample To Borland Enterprise Server 5.2

Deploy the EAR file using `iastool` or the console. Note that you may have to include the JDO library in your stub generation process. Also be sure that you have followed the JCA instructions for BES as well as editing the `samples.properties` to point to `serial://kodo` JNDI location. You should be able to see the `CarEJB` located in JNDI located at the home classname.

2.4.5. Using The Sample Application

The sample application installs itself into the web layer at the context root of `sample`. By browsing to `http://yourserver:yourport/sample`, you should be presented with a simple list page with no cars. You can edit, add, delete car instances. In addition, you can query on the underlying `Car PersistenceCapable` instance by passing in a JDOQL query into the marked form (such as `model=="Some Model"`).

2.4.6. Sample Architecture

The garage application is a simple enterprise application that demonstrates some of the basic concepts necessary when using Kodo in the enterprise layer.

The core model wraps a stateless session bean facade around a persistence capable instance. Using a session bean provides both a

remote interface for various clients as well as providing a transactional context in which to work (and thus avoiding any explicit transactional code).

This session bean uses the Data Transfer Object pattern to provide the primary communication between the application server and the (remote) client. The `Car` instance will be used as the primary object upon which the client will work.

This model can be easily adapted to using entity beans. See our sample `ejb` directory and `JDOEntityBean` for more details on how to using JDO to power your entity beans.

- `samples/jdo/j2ee/Car.java`: The core of the sample application. This is the persistence capable class that Kodo will use to persist the application data. Instances of this class will also fill the role of data transfer object (DTO) for EJB clients. To accomplish this, `Car` implements `java.io.Serializable` so that remote clients can access cars as parameters and return values from the EJB.
- `samples/jdo/j2ee/package.jdo`: The JDO metadata file for the package. Note that some appservers have a problem with the way we include an internal DTD. If you see a lot of illegal/malformed character exceptions, uncomment out the DTD declaration in this file.
- `samples/jdo/j2ee/SampleUtilities.java`: This is a simple facade to aggregate some core JDO functionality into some static methods. By placing all of the functionality into a single facade class, we can reduce code maintenance, as well as having the added advantage of being able to access Kodo from other portions of the application such as a servlet or JSP (though this functionality is not demonstrated in this sample). In addition, some simple utility functions such as JNDI helper methods are also in this class.
- `samples/jdo/j2ee/ejb/Car*.java`: The source for the `CarEJB` session bean. Clients can use the `CarHome` and `CarRemote` interfaces to find, manipulate, and persist changes to `Car` transfer object instances. By using J2EE transactional features, the implementation code in `CarBean.java` can be focused almost entirely upon business and persistence logic without worrying about transactions.
- `samples/jdo/j2ee/jsp/*.jsp`: The web presentation client. These JSPs are not aware of JDO; they simply use the `CarEJB` session bean and the `Car` transfer object to do all the work.
- `samples/jdo/j2ee/resources/*`: Files required to deploy to the various appservers, including J2EE deployment descriptors, WAR/EJB/EAR descriptors, as well as appserver specific files.
- `samples/jdo/j2ee/build.xml`: A simple Ant build file to help in creating a J2EE EAR file.
- `samples/jdo/j2ee/samples.properties`: A simple `.properties` file to tailor the sample application to your particular appserver and installation.

2.4.7. Code Notes and J2EE Tips

1. Persistence capable instances are excellent candidates for the Data Transfer Object Pattern. This pattern attempts to reduce network load, as well as group business logic into concise units. For example, `CarBean.edit` allows you to ensure that all values are correct before committing a transaction, instead of sequentially calling getters and setters on the session facade. This is especially true when using RMI such as from a Swing based application connecting to an application server.

`CarEJB` works as a session bean facade to demarcate transactions, provide finder methods, and encapsulate complex business logic at the server level.

2. Persistent instances using datastore identity lose all identity upon serialization (which happens when they are returned from an EJB method). While there are numerous ways to solve this problem, the sample uses the `clientId` field of `Car` to store the stringified datastore id. *Note that this field is not persistent.* This field ensures that the client and EJB always share the same identity for a given car instance.

Note

Note that this situation is slightly different for application identity. While persistent instances under application-identity still lose their id instance, recreating it from the server side is trivial as the fields on which it is based are still available.

Other ways of solving this problem include:

- Transmitting the object id first or otherwise storing identity with the client:

```
Object oid = dogRemote.findByQuery ("// some query");
Dog dog = dogRemote.getDogForOid (oid);
// changes happen to dog
dogRemote.save (oid, dog);
```

- Wrapper objects that store the persistent instance, object id instance, as well as potentially other application-specific data:

```
public class DogTransferObject
{
    private Dog dog;
    private Object oid;
    private String authentication; // some application specific client data
}
```

3. `PersistenceManager.close` should be called at the end of every EJB method. In addition to ensuring that your code will not attempt to access a closed `PersistenceManager`, it allows the JDO implementation to free up unused resources. Since a `PersistenceManager` is created for every transaction, this can increase the scalability of your application.
4. You should not use `PersistenceManager.currentTransaction` or any `javax.jdo.Transaction` methods. Instead, use the JTA transactional API. `SampleUtilities` includes a helper method to obtain a `UserTransaction` instance from JNDI (see your application server's documentation on the proper JNDI lookup).
5. While serialization of PC instances is relatively straightforward, there are several things to keep in mind:
 - Collections and iterators returned from `Query` instances become closed and inaccessible upon method close. If the client is receiving the results of a `Query`, such as in `CarBean.query`, the results should be transferred to another fresh collection instance before being returned from the server.
 - While "default fetch group" values will always be returned to the client upon serialization, lazily loaded fields will not as the `PersistenceManager` will have been closed before those fields attempt to serialize. One can either simply access those fields before serialization, or one can use the persistence manager's `retrieve` and `retrieveAll` methods to make sure object state is loaded. Note that these methods are not recursive. If you need to go through multiple relations, you must call `retrieve` at each relational depth. This is an intentional limitation in the specification to prevent the entire object graph from being serialized and/or retrieved.
6. It is not necessarily required that you use EJBs and container-managed transactions to demarcate transactions, although that is probably the most common method. In EJBs using bean managed transactions, you can control transactions through either the `javax.jdo.Transaction` or the `javax.transaction.UserTransaction`. Furthermore, outside of EJBs you can access the JDO layer with either transactional API.
7. The Kodo distribution includes source code for some convenient base classes that encapsulate much of the code that is laid out in this example for clarity. `JDOBean`, `JDOSessionBean`, and `JDOEntityBean` include most of the functionality of

SampleUtilities, and handle common EJB interface methods such as `setEntityContext`. To use these classes, we recommend placing Kodo's jars into the system classpath and not into the ear. Ear deployment can cause classloader problems due to the multiple locations that these classes could be loaded from.

8. PersistenceManagers are allocated on a per-Transaction basis. Calling `getPersistenceManager` from the same `PersistenceManagerFactory` within the same EJB method call will always return the same instance.

```
SampleUtilities.getPersistenceManager ()  
    == SampleUtilities.getPersistenceManager (); // will always be true
```

Part 5. Kodo Frequently Asked Questions

Table of Contents

1. Kodo JPA/JDO Frequently Asked Questions	371
1.1. General	371
1.2. Database	373
1.3. Programming with Kodo	375
1.4. How do I ... ?	378
1.5. Common errors	379
1.6. Productivity tools	380
1.7. Performance	380
1.8. Scalability	381
1.9. Application servers	381
1.10. Locking	382
1.11. Transactions	382

Chapter 1. Kodo JPA/JDO Frequently Asked Questions

1.1. General

1.1.1. What is Kodo JPA/JDO?

Kodo is an implementation of the Java Persistence API (JPA) and Java Data Objects (JDO) standards that enables developers to transparently access persistent datastore via the Java programming language.

1.1.2. What is JPA?

JPA stands for Java Persistence API, and is a standard written by Sun Microsystems to provide transparent access to relational stores. A good introduction to JPA can be found at **Chapter 1, Introduction [12]**.

1.1.3. What is JDO?

JDO stands for Java Data Objects, and is a standard written by Sun Microsystems to provide transparent access to a variety of datastores, from relational databases to object databases to plain files. A good introduction to JDO can be found at **Chapter 1, Introduction [173]**.

1.1.4. Is Kodo a database?

No. Kodo provides a means to access an existing database.

1.1.5. Is Kodo an application server?

No, although Kodo can integrate seamlessly with any J2EE 1.3 compliant application server.

1.1.6. Does Kodo require an application server?

No. Kodo can be run without any external managed environment, although it can also be used from within an EJB container, a servlet, or any other managed environment that is J2EE compliant.

1.1.7. What is the difference between JPA and JDBC?

Java Database Connectivity (JDBC) is an API that allows developers to directly access a relational database. JPA is an approach to object persistence that aims to reduce the complexity of designing persistent applications. Kodo JPA/JDO utilizes JDBC to access the relational database.

1.1.8. What is the difference between JDO and JDBC?

Java Database Connectivity (JDBC) is an API that allows developers to directly access a relational database. JDO is a data-store-agnostic approach to object persistence that aims to reduce the complexity of designing persistent applications, and is not constrained to any particular type of datastore. Kodo JPA/JDO utilizes JDBC to access the relational database.

- 1.1.9. What is the difference between JPA and EJB 2?

EJB 2 entity beans are managed distributed components. In contrast, JPA simply provides a transparent means to persist Java objects to a datastore.

- 1.1.1
0. What is the difference between JDO and EJB 2?

Enterprise Java Beans are managed distributed components that handle application-level security and automatic transaction demarcation. In contrast, JDO simply provides a transparent means to persist Java objects to a datastore. EJB and JDO are complimentary technologies; developers can write their EJBs to utilize the transparent persistence provided by JDO, rather than being limited to the restrictions of using the built-in CMP persistence or vendor-specific application server extensions.

- 1.1.1
1. Do I need to know SQL to use Kodo?

No. Kodo completely shields the developer from needing to write or debug SQL statements, although it does allow you to use SQL and JDBC APIs if you want to.

- 1.1.1
2. What standards does Kodo conform to?

Kodo supports the JPA and JDO specifications. Additionally, various parts of Kodo conform to other standards and specifications, including XML, JTA, JCA, JNDI, JDBC, EJB 2, JMX, XA, and J2EE.

- 1.1.1
3. What version of Java does Kodo require?

Kodo JPA/JDO requires JDK 1.5 or higher.

- 1.1.1
4. I have problems or questions about Kodo. Where can I go for help?

The Kodo developer community can be accessed from <http://solarmetric.com/Support/Newsgroups>. Other support resources can be accessed at <http://solarmetric.com/Support>. Also, if you have a maintenance and technical support contract, you can e-mail questions to support@solarmetric.com.

- 1.1.1
5. I think I found a bug in Kodo. Where do I report it?

The first thing you should do is search for any existing bugs in our bug tracking system: <http://bugzilla.solarmetric.com>. It is often the case that someone else may have already reported the bug, and a possible solution or workaround can be found in the existing bug report. If you are confident that your issue has not already been reported, you can report it to the development team by posting on the community newsgroups or sending mail to support@solarmetric.com.

1.2. Database

- 1.2.1. Does Kodo require a database to function?

Kodo does require an existing database against which to operate. However, Kodo ships with a small, open-source, pure-java database called Hypersonic that can be used for development without requiring an existing database installation.

- 1.2.2. What databases does Kodo support?

Kodo has built-in support for all the major databases, including Oracle, Microsoft SQL Server, DB2, Sybase, and Informix. In addition, Kodo provides APIs that allow the developer to adapt Kodo to work with any other database that provides a JDBC-compliant driver. A full list of the supported databases can be found at **Appendix 4, *Supported Databases* [629]**

- 1.2.3. What additional software do I need to use Kodo with my database?

Kodo requires the Java Development Kit. Additionally, if you will be accessing a database other than Hypersonic, Kodo requires a JDBC driver that can communicate with your database. All other libraries needed by Kodo are provided in the Kodo distribution. A list of all the libraries that Kodo requires and uses can be found at **Appendix 7, *Development and Runtime Libraries* [651]**

- 1.2.4. Where can I find the JDBC driver for my application?

JDBC drivers are typically obtained from the database vendor's web site. In addition, there are many third-party companies that provide JDBC drivers for various databases. For a comprehensive list of existing JDBC drivers, see the Sun JDBC driver database at: <http://servlet.java.sun.com/products/jdbc/drivers>.

- 1.2.5. What is the best JDBC driver to use for my application?

We do not usually endorse any specific JDBC driver for a particular database. Provided the driver is truly JDBC compliant, it should work with Kodo without problems. Typically it is a good idea to first look at the JDBC driver that your database vendor provides, since these are often high-quality and free. An advantage of an object persistence API is that you can simply "drop in" a different JDBC driver version and change a few properties, and you can test your application without changing any code. This makes the process of profiling your application's performance with different JDBC drivers a very simple task.

- 1.2.6. What version of JDBC does Kodo use?

Kodo utilizes version 2.0 of the JDBC specification, and requires that a driver be JDBC 2.0 compliant. In addition, Kodo uses the JDBC 2 Standard Extensions, which are provided in the Kodo distribution.

- 1.2.7. Can Kodo integrate with a legacy database schema?

Certainly. Kodo provides a very flexible set of mapping options to be able to integrate your Java object model with almost any relational database schema. In addition, Kodo provides extensible APIs that allow the developer to create their own mappings for those non-standard relational constructs that may not be included with Kodo "out of the box". Furthermore,

Kodo provides a reverse mapping tool, which allows developers to automatically generate a Java object model from an existing schema, which dramatically reduces the amount of time the developer needs to spend manually setting up the mappings. To get started with the reverse mapping tool, see [???](#), [Section 2.3, “Reverse Mapping Tool Tutorial” \[358\]](#).

1.2.8.

I am designing a persistent model from scratch and don't want to deal with creating a database schema. Can Kodo generate a schema for me?

Yes. Kodo allows you to design your application in a object-centric fashion, and can automatically generate a consistent relational schema from your object model. This is ideal for developers who are designing a new application that does not need to integrate with an existing schema. See [Section 7.1, “Forward Mapping” \[489\]](#)

1.2.9.

I have both an existing object model and an existing database schema. Can I use Kodo without making changes to either?

Probably. Kodo's mapping capabilities are quite flexible, and the Kodo mapping tools provide facilities to deal with this "meet-in-the-middle" scenario. See [Section 7.3, “Meet-in-the-Middle Mapping” \[499\]](#) for a more detailed description.

1.2.1

0. Can Kodo generate the DDL for my database?

Yes. Kodo can either issue the DDL directly in order to create or update your database, or it can create a SQL script file. This can be done from the command line or using Ant. See [Section 7.1.2, “Generating DDL SQL” \[493\]](#) for examples.

1.2.1

1. Can a Kodo run against multiple databases?

Yes. Kodo can operate simultaneously against an arbitrary number of databases at the same time. Utilizing XA transactions, you can even ensure transactional consistency across a heterogeneous set of databases when using container managed transactions.

1.2.1

2. Is there any limit to the number of rows Kodo can handle?

There is no limit to the size of table against Kodo can operate, nor is there any limit to the number of tables in the database. Kodo is used in applications that use databases ranging from just a few tables, to databases that have thousands of tables with millions of rows.

1.2.1

3. Can Kodo work with foreign keys?

Kodo can work with both deferred and non-deferred constraints. With non-deferred constraints, Kodo analyzes foreign key dependencies before executing SQL. With deferrable constraints, you should either make deferred the default or extend `DBDictionary`. to set deferred as the default for every connection.

1.2.1

4. How can I change what schema Kodo is using?

Kodo can be configured to look at a subset of schemas, assign a default schema to mapping information, and to ignore the schema altogether. The first is controlled by the [Section 2.8.12, “kodo.jdbc.Schemas” \[415\]](#) option which enumerates the schemas to analyze. The other options are configured at the `DBDictionary` level with the `UseSchemaName` and `DefaultSchemaName` properties (see [Section 4.4, “Database Support” \[428\]](#))

- 1.2.1
5. What do I have to do when using an external `DataSource`?

While Kodo can handle external `DataSource`s, there are certain additional configuration options to be aware of. If your external `DataSource` automatically integrates with the global transaction (such as XA `DataSource`s), you should ensure that Kodo has access to a non-transactional `DataSource`: **Section 4.2.1, “Managed and XA `DataSource`s” [426]**

- 1.2.1
6. How has the schema tool changed since version 2.x?

The schema tool has been redesigned to only manage the schema through XML representations of the schema. The *mapping tool* now manages the mapping between your classes and database. The mapping tool can optionally use the schema tool to synchronize the database schema and create any missing database objects. For further information on the mapping tool, see **Section 7.1, “Forward Mapping” [489]**

1.3. Programming with Kodo

- 1.3.1.
How long will it take me to learn JPA?

JPA is designed to be extremely simple. You can view the entire Java Persistence API at **<http://java.sun.com/javaee/5/docs/api/index.html>**.

- 1.3.2.
How long will it take me to learn the JDO APIs?

The JDO API is designed to be extremely simple. You can view the entire JDO API at **[../jdo-javadoc/index.html](http://java.sun.com/jdo-javadoc/index.html)**.

- 1.3.3.
What standard APIs do I need to be familiar with to use Kodo?

Aside from the Java Persistence and JDO APIs, you do not need to have any expertise in any APIs aside from the basic standard ones that the Java core library provides.

- 1.3.4.
What is the fastest way to get going with Kodo?

The Kodo tutorial provides a good introduction to developing a small application. See **Section 1.2, “Kodo JPA Tutorial” [329]**, **Section 2.2, “Kodo JDO Tutorial” [346]**.

- 1.3.5.
Do you provide any example applications using Kodo?

Kodo ships with numerous sample applications that can be adapted for your needs. For an overview of the samples that come with the Kodo distribution, see **Chapter 1, *Kodo Sample Code* [618]**

- 1.3.6.
How do I issue queries to the database using Kodo?

JPA specifies a query language called JPQL, which allows queries to be written in an object-centric, SQL-like syntax. See

Chapter 10, *JPA Query* [82]. JPA also offers the ability to directly execute SQL queries against the database. See **Chapter 11, *SQL Queries* [116].**

JDO specifies a query language called JDOQL, which allows queries to be written in an object-centric, Java-like syntax. See **Chapter 11, *Query* [232].** JDO also offers the ability to directly execute SQL queries against the database. See **Chapter 17, *SQL Queries* [320].**

1.3.7. How much more programming do I need to do to use Kodo?

One of the goals of Kodo's transparent persistence is to make the persistence code in your application as minimal and unintrusive as possible. Typically, you will only need to write in the transaction demarcation, object queries, and the addition of root objects to the database.

1.3.8. What advantages does Kodo have over other persistence APIs?

Kodo is much less intrusive than other persistence APIs, in that your code does not need to be constantly "polluted" with additional code to do things like traversing relations. In addition, Kodo's bytecode enhancement allows performance and scalability advantages that cannot be matched by other reflection-based persistence architectures, such as declarative "fetch groups", automatic change detection, and transparent relation traversal.

1.3.9. Can Kodo be used in conjunction with other applications that operate against the same database?

Yes. Kodo does not require that it have exclusive read or write access to your database. The only restriction is that some optimistic version strategies may need to be respected by other applications.

1.3.1
0. Does Kodo provide extensions to JPA?

As well as providing complete support for the core Java Persistence API set, Kodo does provide extensions for some advanced operations. The Reference Guide details these extensions.

1.3.1
1. Does Kodo provide extensions to the JDO API?

As well as providing complete support for the core JDO API, Kodo does provide API extensions for some advanced or JDB-specific operations. The Reference Guide details these extensions.

1.3.1
2. How do I avoid vendor lock-in when using Kodo?

To avoid tying your application to any particular vendor, you should avoid using any non-standard API extensions. This will ensure that you can write your application in a way that will run in exactly the same way with any spec-compliant software.

1.3.1
3. Is the application that I write in Kodo portable to other JPA or JDO vendors?

Yes, provided that the vendor's implementation is truly compliant with the specification.

1.3.1
4. How does Kodo interact with the data access/transfer pattern (DAO / DTO)?

While you can wrap Kodo in the DAO pattern, most users find it easier to use Kodo APIs directly. Kodo provides much of DAO's functionality in a standardized and easy to use API set.

- 1.3.1
5. How does Kodo affect the build process of my application?

When using Java 5 or higher, Kodo does not require any changes to your build process. Under previous Java versions, or if you choose not to take advantage of Kodo's Java 5 runtime enhancement, you will have to run the Kodo enhancer after compiling your persistent classes. See [Section 5.2, "Enhancement" \[452\]](#)

- 1.3.1
6. What is bytecode enhancement?

Bytecode enhancement involves changing your compiled classes to mediate access to all the fields that are marked as persistent. See [Section 5.2, "Enhancement" \[452\]](#)

- 1.3.1
7. How can I debug enhanced persistent classes?

Enhanced classes retain their line number tables. This means that lines in stack traces will match those of your original Java source file. Furthermore, enhanced classes can be used by debuggers in exactly the same way as unenhanced classes.

- 1.3.1
8. Can I use Kodo without having to enhance my classes?

Kodo does not require bytecode enhancement to function, although not using enhancement involves writing all your persistent classes to implement the `kodo.enhance.PersistenceCapable` interface. As well as removing some of the transparency of Kodo, it makes the process considerably more complex.

- 1.3.1
9. What is the differences between data caching and query caching?

Data caching caches data from the database. Basically, it caches your persistent objects. Query caching caches the identifiers of query result objects. The next time you run the same query, Kodo uses the cached identifiers to look up the result objects from the data cache, rather than running a database query.

So, an example of where these are different are if you did a query for "salary > 100" and then a query for "salary > 1000", the second query would be run against the database (it couldn't use the prior cached query), but all of the retrieved objects would be in the data cache, so the data for them wouldn't need to be retrieved.

- 1.3.2
0. When using the mappingtool to create a schema for me, why does Kodo create a column called NAME0 for a field called name?

When Kodo automatically generates a column name for a field, it ensures that the name fits within the limitations of your database. This means that a long field name might be truncated, and that fields whose names are database keywords (such as `name`) will have a 0 appended to the column name.

If the auto-generated column names bother you, you can always manually edit your mapping information to control exactly what your schema should look like, or plug in your own mapping defaults through the `kodo.jdbc.MappingDefaults`.

1.4. How do I ... ?

- 1.4.1. I would like to have dynamic control over fetch groups at runtime. Is this possible?

Yes, using custom fetch groups, or simply by adding the fields you want fetched to Kodo's internal fetch plan. See [Section 5.6, “Fetch Groups” \[469\]](#)

- 1.4.2. I want to decouple a persistent instance from its `PersistenceManager` and `Transaction`. How can I do this?

There are a number of ways to decouple an instance from the `PersistenceManager`. The simplest is to just call `makeTransient` on the object, which will decouple the object (but not related objects) from the `PersistenceManager`. If you want to decouple the object as well as all its relations, you can serialize the object and then deserialize it, but this will traverse the entire object graph, which could potentially draw down the entire contents of the database, with catastrophic memory and performance consequences. The third way is to use JDO's `attach` and `detach` APIs, which allows an object to be detached (and then possibly serialized later). For information about detaching instances, see [Section 8.7, “Detach and Attach Functionality” \[219\]](#).

- 1.4.3. Can I see the SQL that Kodo is issuing to the database?

Yes. You can enable the SQL logging channel to see all the SQL statements that are sent from the database. You can also enable the JDBC channel to see most of the JDBC operations (such as commit and rollback operations) that are executed. For details on logging configuration, see [Chapter 3, Logging \[417\]](#)

- 1.4.4. Can Kodo use my existing logging framework instead of its own?

Yes. Kodo has built-in support for Log4J and the Apache Commons Logging frameworks. In turn, the Commons Logging framework can be configured to use JDK 1.4 `java.util.logging`. Additionally, it is possible to plug in your own logging implementation to override Kodo's default behavior. For details on logging configuration, see [Chapter 3, Logging \[?\]](#).

- 1.4.5. I would like to execute my queries in raw SQL. Is this possible?

Yes. You can execute queries in raw SQL and get the results back as either persistent objects, value arrays, or data structs.

For executing SQL queries through the JPA, see [Chapter 11, SQL Queries \[116\]](#).

For executing SQL queries through the JDO API, see [Chapter 17, SQL Queries \[320\]](#).

- 1.4.6. How do I issue a query against a `Date` field?

You need to use a parameter to the query. For details, see [Section 11.2, “JDOQL” \[234\]](#). An example of this is:

Example 1.1. Issuing a query against a Date field

```
Query query = pm.newQuery (Magazine.class, "publishedDate < :now");
```

```
List results = (List) query.execute (new Date ());
```

- 1.4.7. When is the object id available to be read when creating a new persistent instance?

When you first ask for the object id (via standard APIs or by reading a primary key field), or on flush - whatever happens first. In fact, if the identity of your object depends on auto-increment columns, asking for the object id can cause an implicit flush to get the database-generated primary key value(s).

Once you have flushed or retrieved the id of an object, that id is permanent. If the object uses application identity, attempting to change any primary key fields will cause an exception.

- 1.4.8. How do I do query-by-example in Kodo?

You can provide a template object for Kodo to compare to by using a non-persistent parameter to a query.

See **Example 11.8, “Query By Example” [238]** for details on how this works in JDO.

1.5. Common errors

- 1.5.1. When using application identity, what can cause strange behavior like objects not being found?

The most common cause of problems like these when using application identity is failure for the application identity class to properly override the `equals` and `hashCode` methods so that identity objects with equivalent primary key values are considered equal.

- 1.5.2. What causes connection errors when returning persistent instances from a session bean?

A common cause of this problem is due to the EJB container serializing the instances that are being returned from the EJB. The serialization process happens at a point in the EJB lifecycle where the current status of the transaction is undefined. Since serialization may result in unloaded relations being traversed, Kodo will try to obtain a JDBC Connection to perform the traversal, and the application server may then disallow the connection access due to an invalid transaction status. The simplest solution to this is to either make the object to be returned transient, or return a detached instance, or manually perform the serialization before returning from the EJB method.

- 1.5.3. Why do my relations get lost after I commit?

The problem may be that you have defined a bidirectional relation, but you did not set both sides of the relation. Kodo does not perform any "magic" to keep relations consistent by default; the application must always ensure that the Java object model is consistent. You can, however, configure Kodo to manage bidirectional relations for you as described in **Section 5.4, “Managed Inverses” [458]**

1.6. Productivity tools

- 1.6.1. Can Kodo be used with Apache Ant?

Apache Ant is a very popular build tool for Java projects. Kodo has full support for Ant by providing custom Ant tasks for all of the Kodo development tasks. See [Section 14.1, “Apache Ant” \[607\]](#)

1.7. Performance

- 1.7.1. Does Kodo use any caching?

Yes. Kodo has two levels of caching. The first is a per-session cache that is mandated by the specification. Kodo also provides a level 2 data cache that can be shared by multiple sessions and has the capability of synchronizing across a distributed system. See [Section 10.1, “Data Cache” \[560\]](#)

- 1.7.2. Is Kodo faster than JDBC?

A Kodo application will typically outperform a generic JDBC application, since Kodo is able to efficiently batch together like operations at commit time, eliminate redundant SQL, and perform sophisticated caching. Since Kodo runs atop a JDBC driver, it will never be able to communicate with the database any faster than the JDBC driver can. However, for any operations beyond the most simplistic JDBC program, Kodo will dramatically increase the performance of the application beyond what is possible using raw JDBC.

- 1.7.3. Is Kodo faster than EJB 2 Entity Beans?

It is only meaningful to compare the performance of Kodo with that of CMP Entity Beans. While the performance of CMP beans varies depending on the application server being used, they will almost always be slower than using plain Java objects, since plain objects does not incur the heavy method invocation overhead that penalizes all EJBs.

- 1.7.4. How can I speed up my Kodo application?

Kodo provides a wide variety of configuration options and API extensions to fine tune your applications performance. For an overview of common optimization techniques, see [Chapter 15, *Optimization Guidelines* \[612\]](#)

- 1.7.5. How can I profile the performance of my application?

Kodo provides a sophisticated set of management tools to analyze the behavior and performance of your application. See [Chapter 12, *Management and Monitoring* \[591\]](#) and [Chapter 13, *Profiling* \[603\]](#) Also, Kodo can be used in all popular profiling applications such as *OptimizeIt* or *JProbe*.

- 1.7.6. Can I customize the fields that Kodo loads when an object is first instantiated from the database?

Yes. Kodo allows the definition of custom fetch groups, which enable the application to dynamically specify which fields

to instantiate eagerly. See [Section 5.6, “Fetch Groups” \[469\]](#)

- 1.7.7. When I traverse a relation, Kodo issues a separate statement to the database. How can I ensure that the relation is always loaded immediately?

Relations, like any other fields, can be added to the default fetch group, which will cause Kodo to attempt to efficiently traverse the relation when the owning persistent instance is first instantiated from the database. See [Section 5.7, “Eager Fetching” \[473\]](#)

- 1.7.8. Does Kodo utilize connection pooling?

Kodo provides its own built-in connection pooling framework. Additionally, Kodo can be integrated with any third-party connection pool that implements the `javax.sql.DataSource` interface (including the connection pools that are provided with all known application servers). See [Section 4.1, “Using the Kodo DataSource” \[423\]](#) and [Section 4.2, “Using a Third-Party DataSource” \[426\]](#)

1.8. Scalability

- 1.8.1. Does Kodo support load balancing?

Yes, you can easily use Kodo in conjunction with a server farm or cluster, with a load balancer in front of the farm. Kodo provides support for synchronizing its caches across JVMs, so you can use Kodo's data cache when in a clustered environment as well.

- 1.8.2. I have to process millions of records, but my servers do not have enough memory to hold all of the records in memory at the same time. Can Kodo handle doing that?

Yes. By default, Kodo eagerly traverses all result sets. However, you can easily configure Kodo to lazily traverse result sets and to release hard references to traversed objects. See [Section 4.10, “Large Result Sets” \[440\]](#)

1.9. Application servers

- 1.9.1. Can Kodo be run inside an application server?

Kodo can be used in any J2EE compliant application server. Kodo integrates with managed environments (such as application servers) in a variety of ways, from synchronization with container managed transactions to support for accessing `DataSources` from JNDI.

- 1.9.2. Which application servers does Kodo support?

Kodo supports any J2EE compliant application server. For ease of configuration and deployment, Kodo recommends (but does not require) using an application server that supports the Java Connector Architecture (JCA). Kodo has been tested

with most popular application servers such as JBoss, BEA Weblogic, IBM Websphere, SunONE, Macromedia JRun, and Borland Enterprise Server. See **Chapter 8, Deployment** [527]

- 1.9.3. Can I integrate Kodo's transactions with the application server's transaction?

Yes. Kodo can automatically integrate with your application server's managed transactions. See **Section 8.2, "Integrating with the Transaction Manager"** [535]

- 1.9.4. Can I use Kodo to implement my bean-managed persistence entity EJBs?

It is possible to use Kodo to implement bean managed persistence (BMP) entity EJBs. However, doing so will introduce the performance penalties incurred by entity beans. The recommended pattern is to use session beans to perform fine-grained persistence operations with Kodo. To get started with using Kodo with EJB 2, see **Section 1.3, "J2EE Tutorial"** [342], **Section 2.4, "J2EE Tutorial"** [362].

1.10. Locking

- 1.10. 1. What types of locking does Kodo provide?

Kodo provides support for both datastore locking and optimistic. Datastore locking is implemented using pessimistic database operations that acquire locks on all objects that are retrieved in a transaction. Optimistic locking, on the other hand, will verify that no other transaction has modified any of the objects at commit time, and throw an exception if the object has been changed. Advanced users can get more fine-grained control over locking using Kodo's object locking APIs. See **Section 9.4, "Object Locking"** [542]

- 1.10. 2. Which kind of locking should I use for my application?

The answer depends greatly on the type of application you are developing. The advantage of using pessimistic locking is that the application does not need to worry about any locking violations at commit time, but at the cost of potentially introducing serious locking contention into your application. The advantage of using optimistic locking is that it can be much faster and makes more efficient use of database connections, since it does not necessarily need to hold open a single connection for the duration of a transaction. The primary disadvantage of optimistic locking is that the application must catch lock violation exceptions at commit time and take the appropriate actions upon failure (such as re-trying the operation, or telling the user a violation has occurred). Advanced users who want something between these extremes can use Kodo's fine-grained object locking APIs, detailed in **Section 9.4, "Object Locking"** [542]

- 1.10. 3. Why do I get optimistic exceptions even though I am using pessimistic transactions?

If your objects have an optimistic version indicator, then Kodo will still perform optimistic locking logic even if you are using pessimistic transactions. This is so that you can mix optimistic and pessimistic transactions over the life cycle of your application.

1.11. Transactions

1.11.

1. How does Kodo use transactions?

The JPA specification defines a `javax.persistence.EntityTransaction` interface that is similar to a JTA transaction. The JPA application will begin a transaction before making changes to objects, and then commit (or rollback) the transaction once the operation is complete. JPA does not mandate any specific transaction demarcation boundaries; the application is free to begin and end transactions in the way that the developer deems suitable. See **Chapter 9, *Transaction* [79]**.

The JDO specification defines a `javax.jdo.Transaction` interface that is similar to a JTA transaction. The JDO application will begin a transaction before making changes to objects, and then commit (or rollback) the transaction once the operation is complete. JDO does not mandate any specific transaction demarcation boundaries; the application is free to begin and end transactions in the way that the developer deems suitable. See **Chapter 9, *Transaction* [226]**.

Kodo can also integrate with your application server's managed transactions.

1.11.

2. Does a Kodo transaction translate directly to a database transaction?

Not necessarily. For optimistic transactions, Kodo will typically only begin a database transaction when the Kodo transaction is committed, or if changes are manually flushed to the database.

Part 6. Kodo JPA/JDO Reference Guide

Table of Contents

1. Introduction	392
1.1. Intended Audience	392
2. Configuration	393
2.1. Introduction	393
2.2. License Configuration	393
2.3. Runtime Configuration	393
2.4. Command Line Configuration	394
2.4.1. Code Formatting	395
2.5. Plugin Configuration	395
2.6. JDO Standard Properties	397
2.6.1. javax.jdo.PersistenceManagerFactoryClass	397
2.7. Kodo Properties	397
2.7.1. kodo.AggregateListeners	397
2.7.2. kodo.AutoClear	397
2.7.3. kodo.AutoDetach	398
2.7.4. kodo.BrokerFactory	398
2.7.5. kodo.BrokerImpl	398
2.7.6. kodo.ClassResolver	398
2.7.7. kodo.Compatibility	398
2.7.8. kodo.ConnectionDriverName	399
2.7.9. kodo.Connection2DriverName	399
2.7.10. kodo.ConnectionFactory	399
2.7.11. kodo.ConnectionFactory2	399
2.7.12. kodo.ConnectionFactoryName	400
2.7.13. kodo.ConnectionFactory2Name	400
2.7.14. kodo.ConnectionFactoryMode	400
2.7.15. kodo.ConnectionFactoryProperties	400
2.7.16. kodo.ConnectionFactory2Properties	401
2.7.17. kodo.ConnectionPassword	401
2.7.18. kodo.Connection2Password	401
2.7.19. kodo.ConnectionProperties	401
2.7.20. kodo.Connection2Properties	401
2.7.21. kodo.ConnectionURL	402
2.7.22. kodo.Connection2URL	402
2.7.23. kodo.ConnectionUserName	402
2.7.24. kodo.Connection2UserName	402
2.7.25. kodo.ConnectionRetainMode	403
2.7.26. kodo.DataCache	403
2.7.27. kodo.DataCacheTimeout	403
2.7.28. kodo.DetachState	403
2.7.29. kodo.DynamicDataStructs	404
2.7.30. kodo.FetchBatchSize	404
2.7.31. kodo.FetchGroups	404
2.7.32. kodo.FilterListeners	404
2.7.33. kodo.FlushBeforeQueries	405
2.7.34. kodo.IgnoreChanges	405
2.7.35. kodo.InverseManager	405
2.7.36. kodo.LicenseKey	405
2.7.37. kodo.LockManager	406
2.7.38. kodo.LockTimeout	406
2.7.39. kodo.Log	406
2.7.40. kodo.ManagedRuntime	406
2.7.41. kodo.ManagementConfiguration	407

2.7.42. kodo.Mapping	407
2.7.43. kodo.MetaDataFactory	407
2.7.44. kodo.Multithreaded	407
2.7.45. kodo.Optimistic	408
2.7.46. kodo.OrphanedKeyAction	408
2.7.47. kodo.NontransactionalRead	408
2.7.48. kodo.NontransactionalWrite	408
2.7.49. kodo.PersistenceServer	408
2.7.50. kodo.ProxyManager	409
2.7.51. kodo.QueryCache	409
2.7.52. kodo.QueryCompilationCache	409
2.7.53. kodo.ReadLockLevel	409
2.7.54. kodo.RemoteCommitProvider	410
2.7.55. kodo.RestoreState	410
2.7.56. kodo.RetainState	410
2.7.57. kodo.RetryClassRegistration	410
2.7.58. kodo.SavepointManager	411
2.7.59. kodo.Sequence	411
2.7.60. kodo.TransactionMode	411
2.7.61. kodo.WriteLockLevel	411
2.8. Kodo JDBC Properties	412
2.8.1. kodo.jdbc.ConnectionDecorators	412
2.8.2. kodo.jdbc.DBDictionary	412
2.8.3. kodo.jdbc.EagerFetchMode	412
2.8.4. kodo.jdbc.FetchDirection	413
2.8.5. kodo.jdbc.JDBCListeners	413
2.8.6. kodo.jdbc.LRSSize	413
2.8.7. kodo.jdbc.MappingDefaults	413
2.8.8. kodo.jdbc.MappingFactory	414
2.8.9. kodo.jdbc.ResultSetType	414
2.8.10. kodo.jdbc.Schema	414
2.8.11. kodo.jdbc.SchemaFactory	414
2.8.12. kodo.jdbc.Schemas	415
2.8.13. kodo.jdbc.SubclassFetchMode	415
2.8.14. kodo.jdbc.SynchronizeMappings	415
2.8.15. kodo.jdbc.TransactionIsolation	415
2.8.16. kodo.jdbc.UpdateManager	416
3. Logging	417
3.1. Logging Channels	417
3.2. Kodo Logging	418
3.3. Disabling Logging	419
3.4. Log4J	420
3.5. Apache Commons Logging	420
3.5.1. JDK 1.4 java.util.logging	420
3.6. Custom Log	421
4. JDBC	423
4.1. Using the Kodo DataSource	423
4.2. Using a Third-Party DataSource	426
4.2.1. Managed and XA DataSources	426
4.3. Runtime Access to DataSource	427
4.4. Database Support	428
4.4.1. DBDictionary Properties	430
4.4.2. MySQLDictionary Properties	435
4.4.3. OracleDictionary Properties	435
4.5. Setting the Transaction Isolation	436
4.6. Setting the SQL Join Syntax	436
4.7. Accessing Multiple Databases	438
4.8. Configuring the Use of JDBC Connections	438
4.9. Statement Batching	440

4.10. Large Result Sets	440
4.11. Default Schema	443
4.12. Schema Reflection	443
4.12.1. Schemas List	443
4.12.2. Schema Factory	443
4.13. Schema Tool	444
4.14. XML Schema Format	447
4.15. The SQLLine Utility	449
5. Persistent Classes	452
5.1. Persistent Class List	452
5.2. Enhancement	452
5.2.1. Enhancing at Build Time	453
5.2.2. Enhancing JPA Entities on Deployment	454
5.2.3. Enhancing at Runtime	454
5.2.4. Serializing Enhanced Types	455
5.3. Object Identity	455
5.3.1. Datastore Identity Objects	456
5.3.2. Application Identity Tool	456
5.3.3. Autoassign / Identity Strategy Caveats	458
5.4. Managed Inverses	458
5.5. Persistent Fields	460
5.5.1. Restoring State	460
5.5.2. Typing and Ordering	460
5.5.3. Calendar Fields and TimeZones	461
5.5.4. Proxies	461
5.5.4.1. Smart Proxies	461
5.5.4.2. Large Result Set Proxies	461
5.5.4.3. Custom Proxies	463
5.5.5. Externalization	464
5.5.5.1. External Values	468
5.6. Fetch Groups	469
5.6.1. Custom Fetch Groups	469
5.6.2. Custom Fetch Group Configuration	470
5.6.3. Per-field Fetch Configuration	471
5.6.4. Implementation Notes	472
5.7. Eager Fetching	473
5.7.1. Configuring Eager Fetching	474
5.7.2. Eager Fetching Considerations and Limitations	475
5.8. Lock Groups	476
5.8.1. Lock Groups and Subclasses	477
5.8.2. Lock Group Mapping	479
6. Metadata	480
6.1. Generating Default JDO Metadata	480
6.2. Metadata Factory	480
6.3. Additional JPA Metadata	481
6.3.1. Datastore Identity	481
6.3.2. Surrogate Version	482
6.3.3. Persistent Field Values	482
6.3.4. Persistent Collection Fields	482
6.3.5. Persistent Map Fields	483
6.4. Metadata Extensions	483
6.4.1. Class Extensions	483
6.4.1.1. Fetch Groups	483
6.4.1.2. Data Cache	483
6.4.1.3. Detached State	484
6.4.1.4. Lock Groups	484
6.4.1.5. Auditable	485
6.4.2. Field Extensions	485
6.4.2.1. Dependent	485

6.4.2.2. LRS	485
6.4.2.3. Order-By	485
6.4.2.4. Inverse-Logical	486
6.4.2.5. Lock Group	486
6.4.2.6. Read-Only	486
6.4.2.7. Type	486
6.4.2.8. Externalizer	487
6.4.2.9. Factory	487
6.4.2.10. External Values	488
6.4.3. Example	488
7. Mapping	489
7.1. Forward Mapping	489
7.1.1. Using the Mapping Tool	491
7.1.2. Generating DDL SQL	493
7.1.3. JDO Forward Mapping Hints	494
7.1.4. Runtime Forward Mapping	495
7.2. Reverse Mapping	495
7.2.1. Customizing Reverse Mapping	498
7.3. Meet-in-the-Middle Mapping	499
7.4. Mapping Defaults	500
7.5. Mapping Factory	503
7.5.1. Importing and Exporting Mapping Data	505
7.6. Non-Standard Joins	506
7.7. Additional JPA Mappings	508
7.7.1. Datastore Identity Mapping	508
7.7.2. Surrogate Version Mapping	509
7.7.3. Multi-Column Mappings	510
7.7.4. Join Column Attribute Targets	510
7.7.5. Embedded Mapping	510
7.7.6. Collections	512
7.7.6.1. Container Table	512
7.7.6.2. Element Columns	512
7.7.6.3. Element Join Columns	512
7.7.6.4. Element Embedded Mapping	512
7.7.6.5. Order Column	513
7.7.6.6. Examples	513
7.7.7. One-Sided One-Many Mapping	514
7.7.8. Maps	515
7.7.8.1. Key Columns	515
7.7.8.2. Key Join Columns	516
7.7.8.3. Key Embedded Mapping	516
7.7.8.4. Examples	516
7.7.9. Indexes and Constraints	517
7.7.9.1. Indexes	517
7.7.9.2. Foreign Keys	517
7.7.9.3. Unique Constraints	518
7.7.9.4. Examples	518
7.8. Mapping Limitations	519
7.8.1. Table Per Class	519
7.9. Mapping Extensions	519
7.9.1. Class Extensions	520
7.9.1.1. Subclass Fetch Mode	520
7.9.1.2. Strategy	520
7.9.1.3. Discriminator Strategy	520
7.9.1.4. Version Strategy	520
7.9.2. Field Extensions	520
7.9.2.1. Eager Fetch Mode	520
7.9.2.2. Nonpolymorphic	520
7.9.2.3. Class Criteria	521

7.9.2.4. Strategy	522
7.9.3. Column Extensions	522
7.9.3.1. insertable	522
7.9.3.2. updatable	522
7.9.3.3. lock-group	522
7.10. Custom Mappings	522
7.10.1. Custom Class Mapping	523
7.10.2. Custom Discriminator and Version Strategies	523
7.10.3. Custom Field Mapping	523
7.10.3.1. Value Handlers	523
7.10.3.2. Field Strategies	523
7.10.3.3. Configuration	524
7.11. Orphaned Keys	525
8. Deployment	527
8.1. Factory Deployment	527
8.1.1. Standalone Deployment	527
8.1.2. EntityManager Injection	527
8.1.3. Kodo JPA JCA Deployment	527
8.1.3.1. WebLogic 9	527
8.1.3.2. JBoss 4.x	528
8.1.4. Kodo JDO JCA Deployment	528
8.1.4.1. WebLogic 6.1 to 7.x	528
8.1.4.2. WebLogic 8.1	529
8.1.4.3. WebLogic 9	530
8.1.4.4. JBoss 3.0	530
8.1.4.5. JBoss 3.2	530
8.1.4.6. JBoss 4.x	531
8.1.4.7. WebSphere 5	531
8.1.4.8. SunONE Application Server 7 / Sun Java Enterprise Server 8-8.1	531
8.1.4.9. Macromedia JRun 4	532
8.1.4.10. Borland Enterprise Server 5.2 - 6.0	532
8.1.5. Non-JCA Application Server Deployment	533
8.2. Integrating with the Transaction Manager	535
8.3. XA Transactions	536
8.3.1. Using Kodo with XA Transactions	536
9. Runtime Extensions	538
9.1. Architecture	538
9.1.1. Broker Customization	539
9.2. JPA Extensions	540
9.2.1. KodoEntityManagerFactory	540
9.2.2. KodoEntityManager	540
9.2.3. KodoQuery	540
9.2.4. Extent	540
9.2.5. StoreCache	541
9.2.6. QueryResultCache	541
9.2.7. FetchPlan	541
9.2.8. KodoPersistence	541
9.3. JDO API Extensions	541
9.3.1. KodoPersistenceManagerFactory	541
9.3.2. KodoPersistenceManager	541
9.3.3. KodoQuery	542
9.3.4. KodoExtent	542
9.3.5. KodoDataStoreCache	542
9.3.6. QueryResultCache	542
9.3.7. KodoFetchPlan	542
9.3.8. KodoJDOHelper	542
9.4. Object Locking	542
9.4.1. Configuring Default Locking	542
9.4.2. Configuring Lock Levels at Runtime	543

9.4.3. Object Locking APIs	544
9.4.4. Lock Manager	546
9.4.5. Rules for Locking Behavior	547
9.4.6. Known Issues and Limitations	547
9.5. Savepoints	548
9.5.1. Using Savepoints	548
9.5.2. Configuring Savepoints	549
9.6. Query Language Extensions	550
9.6.1. Filter Extensions	550
9.6.1.1. Included Filter Extensions	551
9.6.1.2. Developing Custom Filter Extensions	552
9.6.1.3. Configuring Filter Extensions	552
9.6.2. Aggregate Extensions	552
9.6.2.1. Configuring Query Aggregates	552
9.6.3. JDOQL Non-Distinct Results	552
9.6.4. JDOQL Subqueries	553
9.6.4.1. Subquery Parameters, Variables, and Imports	554
9.6.5. MethodQL	554
9.7. Generators	555
9.7.1. Runtime Access	558
9.8. Transaction Events	559
9.9. Non-Relational Stores	559
10. Caching	560
10.1. Data Cache	560
10.1.1. Data Cache Configuration	561
10.1.2. Data Cache Usage	563
10.1.3. Query Cache	566
10.1.4. The Concurrent Data Cache	571
10.1.5. Third-Party Integrations	572
10.1.5.1. Tangosol Integration	572
10.1.5.2. GemStone GemFire Integration	573
10.1.6. Cache Extension	574
10.1.7. Important Notes	575
10.1.8. Known Issues and Limitations	575
10.2. Query Compilation Cache	576
11. Remote and Offline Operation	577
11.1. Detach and Attach	577
11.1.1. Detach Behavior	577
11.1.2. Attach Behavior	577
11.1.3. Defining the Detached Object Graph	578
11.1.3.1. Detached State Field	580
11.1.4. Automatic Detachment	580
11.2. Remote Managers	581
11.2.1. Standalone Persistence Server	582
11.2.2. HTTP Persistence Server	583
11.2.3. Client Managers	583
11.2.4. Data Compression and Filtering	586
11.2.5. Remote Persistence Deployment	587
11.2.6. Remote Transfer Listeners	587
11.3. Remote Event Notification Framework	587
11.3.1. Remote Commit Provider Configuration	587
11.3.1.1. JMS	587
11.3.1.2. TCP	588
11.3.1.3. Common Properties	589
11.3.2. Customization	590
12. Management and Monitoring	591
12.1. Configuration	591
12.1.1. Optional Parameters in Management Group	593
12.1.2. Optional Parameters in Remote Group	594

12.1.3. Optional Parameters in JSR 160 Group	594
12.1.4. Optional Parameters in WebLogic 8.1 Group	595
12.1.5. Configuring Logging for Management / Monitoring	595
12.2. Kodo Management Console	595
12.2.1. Remote Connection	595
12.2.1.1. Connecting to Kodo under WebLogic 8.1	596
12.2.1.2. Connecting to Kodo under JBoss 3.2	596
12.2.1.3. Connecting to Kodo under JBoss 4	597
12.2.2. Using the Kodo Management Console	597
12.2.2.1. JMX Explorer	598
12.2.2.1.1. Executing Operations	598
12.2.2.1.2. Listening to Notifications	598
12.2.2.2. MBean Panel	599
12.2.2.2.1. Notifications / Statistics	599
12.2.2.2.2. Setting Attributes	599
12.3. Accessing the MBeanServer from Code	599
12.4. MBeans	600
12.4.1. Log MBean	600
12.4.2. Kodo Pooling DataSource MBean	600
12.4.3. Prepared Statement Cache MBean	600
12.4.4. Query Cache MBean	600
12.4.5. Data Cache MBean	600
12.4.6. TimeWatch MBean	600
12.4.7. Runtime MBean	601
12.4.8. Profiling MBean	601
13. Profiling	603
13.1. Profiling in an Embedded GUI	603
13.2. Dumping Profiling Data to Disk from a Batch Process	605
13.3. Controlling How the Profiler Obtains Context Information	606
14. Third Party Integration	607
14.1. Apache Ant	607
14.1.1. Common Ant Configuration Options	607
14.1.2. Enhancer Ant Task	609
14.1.3. Application Identity Tool Ant Task	609
14.1.4. Mapping Tool Ant Task	610
14.1.5. Reverse Mapping Tool Ant Task	610
14.1.6. Schema Tool Ant Task	611
14.2. Maven	611
15. Optimization Guidelines	612

Chapter 1. Introduction

Kodo JPA/JDO is a JDBC-based implementation of the JPA and JDO standards for object persistence. This document is a reference for the configuration and use of Kodo JPA/JDO.

1.1. Intended Audience

This document is intended for Kodo JPA/JDO developers. It assumes strong knowledge of Java, familiarity with the eXtensible Markup Language (XML), and an understanding of JPA and JDO persistence. If you are not familiar with JPA or JDO, please read the **JPA Overview** and **JDO Overview** before proceeding. We also strongly recommend taking Kodo's hands-on **tutorials** to get comfortable with Kodo basics.

Certain sections of this guide cover advanced topics such as custom object-relational mapping, enterprise integration, and using Kodo with third-party tools. These sections assume prior experience with the relevant subject.

Chapter 2. Configuration

2.1. Introduction

This chapter describes the Kodo configuration framework. It concludes with descriptions of all the configuration properties recognized by Kodo. You may want to browse these properties now, but it is not necessary. Most of them will be referenced later in the documentation as we explain the various features they apply to.

Kodo JPA/JDO supports both JPA and JDO configuration styles. JPA utilizes the XML format described in **Section 6.1, “persistence.xml” [63]** of the JPA Overview. JDO uses standard Java properties files. Specify your Kodo configuration options in the format native to the specification you are working with. If you are working with both JPA and JDO in the same application, use the configuration format of the specification you use most. JPA and JDO have different standard settings for many properties - for example, JPA looks for entity metadata in annotations, while JDO looks for metadata in `.jdo` files. **The format of your configuration files determines Kodo's default values for these specification-dependent settings.**

Note

You can explicitly declare specification-dependent settings such that a JDO properties file configures Kodo for JPA defaults, or an JPA XML file configures Kodo for JDO defaults. However, it is more robust to use the configuration format native to the desired specification.

2.2. License Configuration

Kodo utilizes BEA's universal licensing system to enable Kodo's feature set and abilities. In order for Kodo to find your license keys, you must ensure that the `license.bea` found in your BEA home directory is at the root of your classpath.

When using Kodo's environment scripts, you should copy `license.bea` to the Kodo installation directory. Alternatively, you can include your BEA home directory in your classpath.

When deploying Kodo at runtime, you should ensure that `license.bea` is available at the root of the application classpath. For example, you may need to alter your application server configuration to include your BEA home directory in the system classpath. For applications re-distributed as a jar, you should include `license.bea` at the root of your re-distributed jar.

Note

To obtain an evaluation BEA license, visit <http://ftpmain.bea.com/download/pub/license/kodo/kodo40/license.bea> to download a license.

2.3. Runtime Configuration

The Kodo runtime includes a comprehensive system of configuration defaults and overrides:

- Kodo first looks for an optional `kodo.xml` resource. Kodo searches for this resource in each top-level directory of your CLASSPATH. Kodo will also find the resource if you place it within a `META-INF` directory in any top-level directory of the CLASSPATH. The `kodo.xml` resource contains property settings in **JPA's XML format**.

If Kodo discovers a `kodo.xml` resource, specification-dependent settings default to JPA values.

- If Kodo does not find a `kodo.xml` resource, it turns to JDO. Kodo JDO searches for an optional `kodo.properties` resource. Place this resource in any top-level directory of the CLASSPATH, or within a `META-INF` directory in any top-level

directory of the CLASSPATH. If found, this resource is parsed as a standard Java properties file.

If Kodo discovers a `kodo.properties` resource, specification-dependent settings default to JDO values.

- You can customize the name or location of the above resource by specifying the correct resource path in the `kodo.properties` System property.
- You can override any value defined in the above resource by setting the System property of the same name to the desired value.
- In JPA, the values in the standard `META-INF/persistence.xml` bootstrapping file used by the `Persistence` class at runtime override the values in the above resource, as well as any System property settings. The Map passed to `Persistence.createEntityManagerFactory` at runtime also overrides previous settings, including properties defined in `persistence.xml`.
- In JDO, the values in the Properties object passed to `JDOHelper.getPersistenceManagerFactory` at runtime override the values in the above resource, as well as any System property settings.
- When using JCA deployment the `config-property` values in your `ra.xml` file override other settings.
- All Kodo command-line tools accept flags that allow you to specify the configuration resource to use, and to override any property. [Section 2.4, “Command Line Configuration” \[394\]](#) describes these flags.

Note

Internally, the Kodo JPA/JDO runtime environment and development tools manipulate property settings through a general `Configuration` interface, and in particular its `KodoConfiguration` and `JDBCConfiguration` subclasses. For advanced customization, Kodo's extended runtime interfaces and its development tools allow you to access these interfaces directly. See the [Javadoc](#) for details.

2.4. Command Line Configuration

Kodo development tools share the same set of configuration defaults and overrides as the runtime system. They also allow you to specify property values on the command line:

- `-properties/-p <configuration file or resource>` : Use the `-properties` flag, or its shorter `-p` form, to specify a configuration file to use. Note that Kodo always searches the default file locations described above, so this flag is only needed when you do not have a default resource in place, or when you wish to override the defaults. The given value can be the path to a file, or the resource name of a file somewhere in the CLASSPATH. Kodo will search the given location as well as the location prefixed by `META-INF/`. Thus, to point a Kodo tool at `META-INF/persistence.xml`, you can use:

```
<tool> -p persistence.xml
```

The settings in the given file override any settings in `kodo.xml` or `kodo.properties`. Note, however, that specification-dependent properties will still default to JPA values if `kodo.xml` exists, or to JDO values if `kodo.properties` exists. If neither exists, then the defaults for these settings will depend on the format of the given file. If it is an XML file, Kodo will default to JPA settings. If it is a properties file, Kodo will default to JDO settings.

- `-<property name> <property value>` : Any configuration property that you can specify in a configuration file can be overridden with a command line flag. The flag name is always the last token of the corresponding property name, with the first letter in either upper or lower case. For example, to override the `kodo.ConnectionUserName` property, you could pass the `-connectionUserName <value>` flag to any tool. Values set this way override both the values in the

configuration file and values set via System properties.

2.4.1. Code Formatting

Some Kodo development tools generate Java code. These tools share a common set of command-line flags for formatting their output to match your coding style. All code formatting flags can begin with either the `codeFormat` or `cf` prefix.

- `-codeFormat./-cf.tabSpaces <spaces>` : The number of spaces that make up a tab, or 0 to use tab characters. Defaults to using tab characters.
- `-codeFormat./-cf.spaceBeforeParen <true/t | false/f>`: Whether or not to place a space before opening parentheses on method calls, if statements, loops, etc. Defaults to `false`.
- `-codeFormat./-cf.spaceInParen <true/t | false/f>`: Whether or not to place a space within parentheses; i.e. `method(arg)`. Defaults to `false`.
- `-codeFormat./-cf.braceOnSameLine <true/t | false/f>`: Whether or not to place opening braces on the same line as the declaration that begins the code block, or on the next line. Defaults to `true`.
- `-codeFormat./-cf.braceAtSameTabLevel <true/t | false/f>`: When the `braceOnSameLine` option is disabled, you can choose whether to place the brace at the same tab level of the contained code. Defaults to `false`.
- `-codeFormat./-cf.scoreBeforeFieldName <true/t | false/f>`: Whether to prefix an underscore to names of private member variables. Defaults to `false`.
- `-codeFormat./-cf.linesBetweenSections <lines>` : The number of lines to skip between sections of code. Defaults to 2.

Example 2.1. Code Formatting with the Reverse Mapping Tool

```
reversemappingtool -cf.spaceBeforeParen true -cf.tabSpaces 4
```

2.5. Plugin Configuration

Because Kodo is a highly customizable environment, many configuration properties relate to the creation and configuration of system plugins. Plugin properties have a syntax very similar to that of Java 5 annotations. They allow you to specify both what class to use for the plugin and how to configure the public fields or bean properties of the instantiated plugin instance. The easiest way to describe the plugin syntax is by example:

Kodo has a pluggable L2 caching mechanism that is controlled by the `kodo.DataCache` configuration property. Suppose that you have created a new class, `com.xyz.MyDataCache`, that you want Kodo to use for caching. You've made instances of `MyDataCache` configurable via two methods, `setCacheSize(int size)` and `setRemoteHost(String host)`. The sample below shows how you would tell Kodo to use an instance of your custom plugin with a max size of 1000 and a remote host of `cacheserver`.

JPA XML format:

```
<property name="kodo.DataCache"
  value="com.xyz.MyDataCache(CacheSize=1000, RemoteHost=cacheserver)"/>
```

JDO properties format:

```
kodo.DataCache: com.xyz.MyDataCache(CacheSize=1000, RemoteHost=cacheserver)
```

As you can see, plugin properties take a class name, followed by a comma-separated list of values for the plugin's public fields or bean properties in parentheses. Kodo will match each named property to a field or setter method in the instantiated plugin instance, and set the field or invoke the method with the given value (after converting the value to the right type, of course). The first letter of the property names can be in either upper or lower case. The following would also have been valid:

```
com.xyz.MyDataCache(cacheSize=1000, remoteHost=cacheserver)
```

If you do not need to pass any property settings to a plugin, you can just name the class to use:

```
com.xyz.MyDataCache
```

Similarly, if the plugin has a default class that you do not want to change, you can simply specify a list of property settings, without a class name. For example, Kodo's query cache companion to the data cache has a default implementation suitable to most users, but you still might want to change the query cache's size. It has a `CacheSize` property for this purpose:

```
CacheSize=1000
```

Finally, many of Kodo's built-in options for plugins have short alias names that you can use in place of the full class name. The data cache property, for example, has an available alias of `true` for the standard cache implementation. The property value simply becomes:

```
true
```

The standard cache implementation class also has a `CacheSize` property, so to use the standard implementation and configure the size, specify:

```
true(CacheSize=1000)
```

The remainder of this chapter reviews the set of configuration properties Kodo recognizes.

2.6. JDO Standard Properties

The JDO specification describes many standard configuration properties, all of which Kodo supports. These properties are covered in [Section 7.2, “PersistenceManagerFactory Properties” \[208\]](#) of the JDO Overview. Kodo also defines its own specification-neutral alternatives to the JDO properties. You may use either Kodo properties or standard JDO properties in your JDO configuration files.

There is one JDO property for which Kodo does not have a native equivalent. The `javax.jdo.PersistenceManagerFactoryClass` property is used by the `JDOHelper` to bootstrap the JDO implementation.

2.6.1. javax.jdo.PersistenceManagerFactoryClass

Property name: `javax.jdo.PersistenceManagerFactoryClass`

Default: -

Description: The name of the `javax.jdo.PersistenceManagerFactory` implementation that `JDOHelper.getPersistenceManagerFactory` should create. For Kodo, this should always be `kodo.jdo.PersistenceManagerFactoryImpl`.

2.7. Kodo Properties

Kodo defines many configuration properties. Most of these properties are provided for advanced users who wish to customize Kodo's behavior; the majority of developers can omit them. The following properties apply to any Kodo back-end, though the given descriptions are tailored to Kodo's default JDBC store.

2.7.1. kodo.AggregateListeners

Property name: `kodo.AggregateListeners`

Configuration API: `kodo.conf.KodoConfiguration.getAggregateListeners`

Resource adaptor config-property: `AggregateListeners`

Default:-

Description: A comma-separated list of plugin strings (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing custom `kodo.jdbc.kernel.exps.JDBCAggregateListeners` to make available to all queries, in addition to the standard set of listeners. See [Section 9.6, “Query Language Extensions” \[550\]](#) for details on aggregate listeners.

2.7.2. kodo.AutoClear

Property name: `kodo.AutoClear`

Configuration API: `kodo.conf.KodoConfiguration.getAutoClear`

Resource adaptor config-property: `AutoClear`

Default: `datastore`

Possible values: `datastore, all`

Description: When to automatically clear instance state: on entering a datastore transaction, or on entering any transaction.

2.7.3. kodo.AutoDetach

Property name: `kodo.AutoDetach`

Configuration API: `kodo.conf.KodoConfiguration.getAutoDetach`

Resource adaptor config-property: `AutoDetach`

Default: -

Possible values: `close, commit, nontx-read`

Description: A comma-separated list of events when managed instances will be automatically detached.

2.7.4. kodo.BrokerFactory

Property name: `kodo.BrokerFactory`

Configuration API: `kodo.conf.KodoConfiguration.getBrokerFactory`

Resource adaptor config-property: `BrokerFactory`

Default: `jdbc`

Possible values: `jdbc, abstractstore, remote`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.kernel.BrokerFactory` type to use.

2.7.5. kodo.BrokerImpl

Property name: `kodo.BrokerImpl`

Configuration API: `kodo.conf.KodoConfiguration.getBrokerImpl`

Resource adaptor config-property: `BrokerImpl`

Default: `default`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.kernel.Broker` type to use at runtime. See [Section 9.1.1, “Broker Customization” \[539\]](#) for details.

2.7.6. kodo.ClassResolver

Property name: `kodo.ClassResolver`

Configuration API: `kodo.conf.KodoConfiguration.getClassResolver`

Resource adaptor config-property: `ClassResolver`

Default: `default`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.util.ClassResolver` implementation to use for class name resolution. You may wish to plug in your own resolver if you have special classloading needs.

2.7.7. kodo.Compatibility

Property name: `kodo.Compatibility`

Configuration API: `kodo.conf.KodoConfiguration.getCompatibility`

Resource adaptor config-property: `Compatibility`

Default: -

Description: Encapsulates options to mimic the behavior of previous Kodo releases. See [Section 6.1, “Compatibility Configuration” \[648\]](#)

2.7.8. `kodo.ConnectionDriverName`

Property name: `kodo.ConnectionDriverName`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionDriverName`

Resource adaptor config-property: `ConnectionDriverName`

Default: -

Description: The full class name of either the JDBC `java.sql.Driver`, or a `javax.sql.DataSource` implementation to use to connect to the database. See [Chapter 4, JDBC \[423\]](#) for details.

2.7.9. `kodo.Connection2DriverName`

Property name: `kodo.Connection2DriverName`

Configuration API: `kodo.conf.KodoConfiguration.getConnection2DriverName`

Resource adaptor config-property: `Connection2DriverName`

Default: -

Description: This property is equivalent to the `kodo.ConnectionDriverName` property described in [Section 2.7.8, “`kodo.ConnectionDriverName`” \[399\]](#), but applies to the alternate connection factory used for unmanaged connections. See [Section 4.2.1, “Managed and XA DataSources” \[426\]](#) for details.

2.7.10. `kodo.ConnectionFactory`

Property name: `kodo.ConnectionFactory`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionFactory`

Resource adaptor config-property: `ConnectionFactory`

Default: -

Description: A `javax.sql.DataSource` to use to connect to the database. See [Chapter 4, JDBC \[423\]](#) for details.

2.7.11. `kodo.ConnectionFactory2`

Property name: `kodo.ConnectionFactory2`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionFactory2`

Resource adaptor config-property: `ConnectionFactory2`

Default: -

Description: An unmanaged `javax.sql.DataSource` to use to connect to the database. See [Chapter 4, JDBC \[423\]](#) for details.

2.7.12. kodo.ConnectionFactoryName

Property name: `kodo.ConnectionFactoryName`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionFactoryName`

Resource adaptor config-property: `ConnectionFactoryName`

Default: -

Description: The JNDI location of a `javax.sql.DataSource` to use to connect to the database. See [Chapter 4, JDBC \[423\]](#) for details.

2.7.13. kodo.ConnectionFactory2Name

Property name: `kodo.ConnectionFactory2Name`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionFactory2Name`

Resource adaptor config-property: `ConnectionFactory2Name`

Default: -

Description: The JNDI location of an unmanaged `javax.sql.DataSource` to use to connect to the database. See [Section 8.3, “XA Transactions” \[536\]](#) for details.

2.7.14. kodo.ConnectionFactoryMode

Property name: `kodo.ConnectionFactoryMode`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionFactoryMode`

Resource adaptor config-property: `ConnectionFactoryMode`

Default: `local`

Possible values: `local`, `managed`

Description: The connection factory mode to use when integrating with the application server's managed transactions. See [Section 4.2.1, “Managed and XA DataSources” \[426\]](#) for details.

2.7.15. kodo.ConnectionFactoryProperties

Property name: `kodo.ConnectionFactoryProperties`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionFactoryProperties`

Resource adaptor config-property: `ConnectionFactoryProperties`

Default: -

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) listing properties for configuration of the

`javax.sql.DataSource` in use. See the [Chapter 4, JDBC \[423\]](#) for details.

2.7.16. `kodo.ConnectionFactory2Properties`

Property name: `kodo.ConnectionFactory2Properties`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionFactory2Properties`

Resource adaptor config-property: `ConnectionFactory2Properties`

Default: -

Description: This property is equivalent to the `kodo.ConnectionFactoryProperties` property described in [Section 2.7.15, “`kodo.ConnectionFactoryProperties`” \[400\]](#), but applies to the alternate connection factory used for unmanaged connections. See [Section 4.2.1, “Managed and XA DataSources” \[426\]](#) for details.

2.7.17. `kodo.ConnectionPassword`

Property name: `kodo.ConnectionPassword`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionPassword`

Resource adaptor config-property: `ConnectionPassword`

Default: -

Description: The password for the user specified in the `ConnectionUserName` property. See [Chapter 4, JDBC \[423\]](#) for details.

2.7.18. `kodo.Connection2Password`

Property name: `kodo.Connection2Password`

Configuration API: `kodo.conf.KodoConfiguration.getConnection2Password`

Resource adaptor config-property: `Connection2Password`

Default: -

Description: This property is equivalent to the `kodo.ConnectionPassword` property described in [Section 2.7.17, “`kodo.ConnectionPassword`” \[401\]](#), but applies to the alternate connection factory used for unmanaged connections. See [Section 4.2.1, “Managed and XA DataSources” \[426\]](#) for details.

2.7.19. `kodo.ConnectionProperties`

Property name: `kodo.ConnectionProperties`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionProperties`

Resource adaptor config-property: `ConnectionProperties`

Default: -

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) listing properties to configure the driver listed in the `ConnectionDriverName` property described below. See [Chapter 4, JDBC \[423\]](#) for details.

2.7.20. `kodo.Connection2Properties`

Property name: `kodo.Connection2Properties`

Configuration API: `kodo.conf.KodoConfiguration.getConnection2Properties`

Resource adaptor config-property: `Connection2Properties`

Default: -

Description: This property is equivalent to the `kodo.ConnectionProperties` property described in [Section 2.7.19](#), “`kodo.ConnectionProperties`” [401], but applies to the alternate connection factory used for unmanaged connections. See [Section 4.2.1](#), “Managed and XA DataSources” [426] for details.

2.7.21. `kodo.ConnectionURL`

Property name: `kodo.ConnectionURL`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionURL`

Resource adaptor config-property: `ConnectionURL`

Default: -

Description: The JDBC URL for the database. See [Chapter 4, JDBC](#) [423] for details.

2.7.22. `kodo.Connection2URL`

Property name: `kodo.Connection2URL`

Configuration API: `kodo.conf.KodoConfiguration.getConnection2URL`

Resource adaptor config-property: `Connection2URL`

Default: -

Description: This property is equivalent to the `kodo.ConnectionURL` property described in [Section 2.7.21](#), “`kodo.ConnectionURL`” [402], but applies to the alternate connection factory used for unmanaged connections. See [Section 4.2.1](#), “Managed and XA DataSources” [426] for details.

2.7.23. `kodo.ConnectionUserName`

Property name: `kodo.ConnectionUserName`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionUserName`

Resource adaptor config-property: `ConnectionUserName`

Default: -

Description: The user name to use when connecting to the database. See the [Chapter 4, JDBC](#) [423] for details.

2.7.24. `kodo.Connection2UserName`

Property name: `kodo.Connection2UserName`

Configuration API: `kodo.conf.KodoConfiguration.getConnection2UserName`

Resource adaptor config-property: `Connection2UserName`

Default: -

Description: This property is equivalent to the `kodo.ConnectionUserName` property described in [Section 2.7.23](#), “`kodo.ConnectionUserName`” [402], but applies to the alternate connection factory used for unmanaged connections. See [Section 4.2.1](#), “Managed and XA DataSources” [426] for details.

2.7.25. `kodo.ConnectionRetainMode`

Property name: `kodo.ConnectionRetainMode`

Configuration API: `kodo.conf.KodoConfiguration.getConnectionRetainMode`

Resource adaptor config-property: `ConnectionRetainMode`

Default: on-demand

Description: Controls how Kodo uses datastore connections. This property can also be specified for individual sessions. See [Section 4.8](#), “Configuring the Use of JDBC Connections” [438] for details.

2.7.26. `kodo.DataCache`

Property name: `kodo.DataCache`

Configuration API: `kodo.conf.KodoConfiguration.getDataCache`

Resource adaptor config-property: `DataCache`

Default: false

Description: A plugin list string (see [Section 2.5](#), “Plugin Configuration” [395]) describing the `kodo.datacache.DataCaches` to use for data caching. See [Section 10.1.1](#), “Data Cache Configuration” [561] for details.

2.7.27. `kodo.DataCacheTimeout`

Property name: `kodo.DataCacheTimeout`

Configuration API: `kodo.conf.KodoConfiguration.getDataCacheTimeout`

Resource adaptor config-property: `DataCacheTimeout`

Default: -1

Description: The number of milliseconds that data in the data cache is valid. Set this to -1 to indicate that data should not expire from the cache. This property can also be specified for individual classes. See [Section 10.1.1](#), “Data Cache Configuration” [561] for details.

2.7.28. `kodo.DetachState`

Property name: `kodo.DetachState`

Configuration API: `kodo.conf.KodoConfiguration.getDetachState`

Resource adaptor config-property: `DetachState`

Default: loaded

Possible values: loaded, fgs, all

Description: Determines which fields are part of the detached graph and related options. For more details, see [Section 11.1.3, “Defining the Detached Object Graph” \[578\]](#)

2.7.29. kodo.DynamicDataStructs

Property name: `kodo.DynamicDataStructs`

Configuration API: `kodo.conf.KodoConfiguration.getDynamicDataStructs`

Resource adaptor config-property: `DynamicDataStructs`

Default: `false`

Description: Whether to dynamically generate customized structs to hold persistent data. Both the Kodo data cache and the remote framework rely on data structs to cache and transfer persistent state. With dynamic structs, Kodo can customize data storage for each class, eliminating the need to generate primitive wrapper objects. This saves memory and speeds up certain runtime operations. The price is a longer warm-up time for the application - generating and loading custom classes into the JVM takes time. Therefore, only set this property to `true` if you have a long-running application where the initial cost of class generation is offset by memory and speed optimization over time.

2.7.30. kodo.FetchBatchSize

Property name: `kodo.FetchBatchSize`

Configuration API: `kodo.conf.KodoConfiguration.getFetchBatchSize`

Resource adaptor config-property: `FetchBatchSize`

Default: `-1`

Description: The number of rows to fetch at once when scrolling through a result set. The fetch size can also be set at runtime. See [Section 4.10, “Large Result Sets” \[440\]](#) for details.

2.7.31. kodo.FetchGroups

Property name: `kodo.FetchGroups`

Configuration API: `kodo.conf.KodoConfiguration.getFetchGroups`

Resource adaptor config-property: `FetchGroups`

Default: `-`

Description: A comma-separated list of fetch group names that are to be loaded when retrieving objects from the datastore. Fetch groups can also be set at runtime. See [Section 5.6, “Fetch Groups” \[469\]](#) for details.

2.7.32. kodo.FilterListeners

Property name: `kodo.FilterListeners`

Configuration API: `kodo.conf.KodoConfiguration.getFilterListeners`

Resource adaptor config-property: `FilterListeners`

Default: `-`

Description: A comma-separated list of full plugin strings (see [Section 2.5, “Plugin Configuration” \[395\]](#)) for custom

`kodo.jdbc.kernel.exps.JDBCFilterListeners` to make available to all queries, in addition to the standard set of listeners. You can also add filter listeners to individual queries. See [Section 9.6, “Query Language Extensions” \[550\]](#) for details.

2.7.33. `kodo.FlushBeforeQueries`

Property name: `kodo.FlushBeforeQueries`

Property name: `kodo.FlushBeforeQueries`

Configuration API: `kodo.conf.KodoConfiguration.getFlushBeforeQueries`

Resource adaptor config-property: `FlushBeforeQueries`

Default: `true`

Description: Whether or not to flush any changes made in the current transaction to the datastore before executing a query. See [Section 4.8, “Configuring the Use of JDBC Connections” \[438\]](#) for details.

2.7.34. `kodo.IgnoreChanges`

Property name: `kodo.IgnoreChanges`

Configuration API: `kodo.conf.KodoConfiguration.getIgnoreChanges`

Resource adaptor config-property: `IgnoreChanges`

Default: `false`

Description: Whether to consider modifications to persistent objects made in the current transaction when evaluating queries. Setting this to `true` allows Kodo to ignore changes and execute the query directly against the datastore. A value of `false` forces Kodo to consider whether the changes in the current transaction affect the query, and if so to either evaluate the query in-memory or flush before running it against the datastore.

2.7.35. `kodo.InverseManager`

Property name: `kodo.InverseManager`

Configuration API: `kodo.conf.KodoConfiguration.getInverseManager`

Resource adaptor config-property: `InverseManager`

Default: `false`

Possible values: `false, true`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing a `kodo.kernel.InverseManager` to use for managing bidirectional relations upon a flush. See [Section 5.4, “Managed Inverses” \[458\]](#) for usage documentation.

2.7.36. `kodo.LicenseKey`

Property name: `kodo.LicenseKey`

Configuration API: `kodo.conf.KodoConfiguration.getLicenseKey`

Resource adaptor config-property: `LicenseKey`

Default: -

Description: This property is deprecated in favor of BEA's universal licensing system. For details on how configure Kodo to use BEA's license file, see [Section 2.2, “License Configuration” \[393\]](#). Kodo will continue to utilize legacy license keys set in this property.

2.7.37. kodo.LockManager

Property name: `kodo.LockManager`

Configuration API: `kodo.conf.KodoConfiguration.getLockManager`

Resource adaptor config-property: `LockManager`

Default: `pessimistic`

Possible values: `none, sjvm, pessimistic`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing a `kodo.kernel.LockManager` to use for acquiring locks on persistent instances during transactions.

2.7.38. kodo.LockTimeout

Property name: `kodo.LockTimeout`

Configuration API: `kodo.conf.KodoConfiguration.getLockTimeout`

Resource adaptor config-property: `LockTimeout`

Default: `-1`

Description: The number of milliseconds to wait for an object lock before throwing an exception, or -1 for no limit. See [Section 9.4, “Object Locking” \[542\]](#) for details.

2.7.39. kodo.Log

Property name: `kodo.Log`

Configuration API: `kodo.conf.KodoConfiguration.getLog`

Resource adaptor config-property: `Log`

Default: `true`

Possible values: `kodo, commons, log4j, none`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing a `com.solarmetric.log.LogFactory` to use for logging. For details on logging, see [Chapter 3, Logging \[417\]](#)

2.7.40. kodo.ManagedRuntime

Property name: `kodo.ManagedRuntime`

Configuration API: `kodo.conf.KodoConfiguration.getManagedRuntime`

Resource adaptor config-property: `ManagedRuntime`

Default: `auto`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.ee.ManagedRuntime` implementation to use for obtaining a reference to the `TransactionManager` in an enterprise environment. See [Section 8.2, “Integrating with the Transaction Manager” \[535\]](#) for details.

2.7.41. kodo.ManagementConfiguration

Property name: `kodo.ManagementConfiguration`

Configuration API: `kodo.conf.KodoConfiguration.getManagementConfiguration`

Resource adaptor config-property: `ManagementConfiguration`

Default: none

Description: Allows for configuration of management and profiling capabilities. For more information, see [Chapter 12, *Management and Monitoring* \[591\]](#)

2.7.42. kodo.Mapping

Property name: `kodo.Mapping`

Configuration API: `kodo.conf.KodoConfiguration.getMapping`

Resource adaptor config-property: `Mapping`

Default: -

Description: The symbolic name of the object-to-datastore mapping to use.

For JDO use, see the equivalent `javax.jdo.option.Mapping` property described in [Section 15.1, “Mapping Metadata Placement” \[260\]](#) of the JDO Overview.

2.7.43. kodo.MetaDataFactory

Property name: `kodo.MetaDataFactory`

Configuration API: `kodo.conf.KodoConfiguration.getMetaDataFactory`

Resource adaptor config-property: `MetaDataFactory`

Default: `jdo`

Possible values: `jdo`, `jpa`, `kodo3`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.meta.MetaDataFactory` to use to store and retrieve metadata for your persistent classes. See [Section 6.2, “Metadata Factory” \[480\]](#) for details.

2.7.44. kodo.Multithreaded

Property name: `kodo.Multithreaded`

Configuration API: `kodo.conf.KodoConfiguration.getMultithreaded`

Resource adaptor config-property: `Multithreaded`

Default: `false`

Description: Whether persistent instances and Kodo components will be accessed by multiple threads at once.

2.7.45. kodo.Optimistic

Property name: `kodo.Optimistic`

Configuration API: `kodo.conf.KodoConfiguration.getOptimistic`

Resource adaptor config-property: `Optimistic`

Default: `true`

Description: Selects between optimistic and pessimistic (datastore) transactional modes.

2.7.46. kodo.OrphanedKeyAction

Property name: `kodo.OrphanedKeyAction`

Configuration API: `kodo.conf.KodoConfiguration.getOrphanedKeyAction`

Resource adaptor config-property: `OrphanedKeyAction`

Default: `log`

Possible values: `log`, `exception`, `none`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing a `kodo.event.OrphanedKeyAction` to invoke when Kodo discovers an orphaned datastore key. See [Section 7.11, “Orphaned Keys” \[525\]](#) for details.

2.7.47. kodo.NontransactionalRead

Property name: `kodo.NontransactionalRead`

Configuration API: `kodo.conf.KodoConfiguration.getNontransactionalRead`

Resource adaptor config-property: `NontransactionalRead`

Default: `true`

Description: Whether the Kodo runtime will allow you to read data outside of a transaction.

2.7.48. kodo.NontransactionalWrite

Property name: `kodo.NontransactionalWrite`

Configuration API: `kodo.conf.KodoConfiguration.getNontransactionalWrite`

Resource adaptor config-property: `NontransactionalWrite`

Default: `false`

Description: Whether you can modify persistent objects and perform persistence operations outside of a transaction. Changes will take effect on the next transaction.

2.7.49. kodo.PersistenceServer

Property name: `kodo.PersistenceServer`

Configuration API: `kodo.conf.KodoConfiguration.getPersistenceServer`

Resource adaptor config-property: `PersistenceServer`

Default: `false`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing whether and how to service remote clients with this factory. See [Section 11.2, “Remote Managers” \[581\]](#) for details.

2.7.50. `kodo.ProxyManager`

Property name: `kodo.ProxyManager`

Configuration API: `kodo.conf.KodoConfiguration.getProxyManager`

Resource adaptor config-property: `ProxyManager`

Default: `default`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing a `kodo.util.ProxyManager` to use for proxying mutable second class objects. See [Section 5.5.4.3, “Custom Proxies” \[463\]](#) for details.

2.7.51. `kodo.QueryCache`

Property name: `kodo.QueryCache`

Configuration API: `kodo.conf.KodoConfiguration.getQueryCache`

Resource adaptor config-property: `QueryCache`

Default: `true`, when the data cache (see [Section 2.7.26, “kodo.DataCache” \[403\]](#)) is also enabled, `false` otherwise.

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.datacache.QueryCache` implementation to use for caching of queries loaded from the data store. See [Section 10.1.3, “Query Cache” \[566\]](#) for details.

2.7.52. `kodo.QueryCompilationCache`

Property name: `kodo.QueryCompilationCache`

Configuration API: `kodo.conf.KodoConfiguration.getQueryCompilationCache`

Resource adaptor config-property: `QueryCompilationCache`

Default: `true`.

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `java.util.Map` to use for caching of data used during query compilation. See [Section 10.2, “Query Compilation Cache” \[576\]](#) for details.

2.7.53. `kodo.ReadLockLevel`

Property name: `kodo.ReadLockLevel`

Configuration API: `kodo.conf.KodoConfiguration.getReadLockLevel`

Resource adaptor config-property: ReadLockLevel

Default: read

Possible values: none, read, write, numeric values for lock-manager specific lock levels

Description: The default level at which to lock objects retrieved during a non-optimistic transaction. Note that for the default JD-BC lock manager, read and write lock levels are equivalent.

2.7.54. kodo.RemoteCommitProvider

Property name: kodo.RemoteCommitProvider

Configuration API: kodo.conf.KodoConfiguration.getRemoteCommitProvider

Resource adaptor config-property: RemoteCommitProvider

Default: -

Description: A plugin string (see Section 2.5, “Plugin Configuration” [395]) describing the `kodo.event.RemoteCommitProvider` implementation to use for distributed event notification. See Section 11.3.1, “Remote Commit Provider Configuration” [587] for more information.

2.7.55. kodo.RestoreState

Property name: kodo.RestoreState

Configuration API: kodo.conf.KodoConfiguration.getRestoreState

Resource adaptor config-property: RestoreState

Default: none

Possible values: none, immutable, all

Description: Whether to restore managed fields to their pre-transaction values when a rollback occurs.

2.7.56. kodo.RetainState

Property name: kodo.RetainState

Configuration API: kodo.conf.KodoConfiguration.getRetainState

Resource adaptor config-property: RetainState

Default: true

Description: Whether persistent fields retain their values on transaction commit.

2.7.57. kodo.RetryClassRegistration

Property name: kodo.RetryClassRegistration

Configuration API: kodo.conf.KodoConfiguration.getRetryClassRegistration

Resource adaptor config-property: RetryClassRegistration

Default: `false`

Description: Controls whether to log a warning and defer registration instead of throwing an exception when a persistent class cannot be fully processed. This property should *only* be used in complex classloader situations where security is preventing Kodo from reading registered classes. Setting this to true unnecessarily may obscure more serious problems.

2.7.58. kodo.SavepointManager

Property name: `kodo.SavepointManager`

Configuration API: `kodo.conf.KodoConfiguration.getSavepointManager`

Resource adaptor config-property: `SavepointManager`

Default: `in-mem`

Possible values: `in-mem, jdbc, oracle`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing a `kodo.kernel.SavepointManager` to use for managing transaction savepoints. See [Section 9.5, “Savepoints” \[548\]](#) for details.

2.7.59. kodo.Sequence

Property name: `kodo.Sequence`

Configuration API: `kodo.conf.KodoConfiguration.getSequence`

Resource adaptor config-property: `Sequence`

Default: `table`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.kernel.Seq` implementation to use for the system sequence. See [Section 9.7, “Generators” \[555\]](#) for more information.

2.7.60. kodo.TransactionMode

Property name: `kodo.TransactionMode`

Configuration API: `kodo.conf.KodoConfiguration.getTransactionMode`

Resource adaptor config-property: `TransactionMode`

Default: `local`

Possible values: `local, managed`

Description: The default transaction mode to use. You can override this setting per-session. See [Section 8.2, “Integrating with the Transaction Manager” \[535\]](#) for details.

2.7.61. kodo.WriteLockLevel

Property name: `kodo.WriteLockLevel`

Configuration API: `kodo.conf.KodoConfiguration.getWriteLockLevel`

Resource adaptor config-property: `WriteLockLevel`

Default: `write`

Possible values: `none`, `read`, `write`, numeric values for lock-manager specific lock levels

Description: The default level at which to lock objects changed during a non-optimistic transaction. Note that for the default JDBC lock manager, `read` and `write` lock levels are equivalent.

2.8. Kodo JDBC Properties

The following properties apply exclusively to the Kodo JDBC back-end.

2.8.1. `kodo.jdbc.ConnectionDecorators`

Property name: `kodo.jdbc.ConnectionDecorators`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getConnectionDecorators`

Resource adaptor config-property: `ConnectionDecorators`

Default: -

Description: A comma-separated list of plugin strings (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing `com.solarmetric.jdbc.ConnectionDecorator` instances to install on the connection factory. These decorators can wrap connections passed from the underlying `DataSource` to add functionality. Kodo will pass all connections through the list of decorators before using them. Note that by default Kodo JPA/JDO employs all of the built-in decorators in the `com.solarmetric.jdbc` package already; you do not need to list them here.

2.8.2. `kodo.jdbc.DBDictionary`

Property name: `kodo.jdbc.DBDictionary`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getDBDictionary`

Resource adaptor config-property: `DBDictionary`

Default: Based on the `kodo.ConnectionURL` `kodo.ConnectionDriverName`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.jdbc.sql.DBDictionary` to use for database interaction. Kodo typically auto-configures the dictionary based on the JDBC URL, but you may have to set this property explicitly if you are using an unrecognized driver, or to plug in your own dictionary for a database Kodo JPA/JDO does not support out-of-the-box. See [Section 4.4, “Database Support” \[428\]](#) for details.

2.8.3. `kodo.jdbc.EagerFetchMode`

Property name: `kodo.jdbc.EagerFetchMode`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getEagerFetchMode`

Resource adaptor config-property: `EagerFetchMode`

Default: `parallel`

Possible values: `parallel`, `join`, `none`

Description: Optimizes how Kodo loads persistent relations. This setting can also be varied at runtime. See [Section 5.7, “Eager Fetching” \[473\]](#) for details.

2.8.4. kodo.jdbc.FetchDirection

Property name: `kodo.jdbc.FetchDirection`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getFetchDirection`

Resource adaptor config-property: `FetchDirection`

Default: `forward`

Possible values: `forward`, `reverse`, `unknown`

Description: The expected order in which query result lists will be accessed. This property can also be varied at runtime. See [Section 4.10, “Large Result Sets” \[440\]](#) for details.

2.8.5. kodo.jdbc.JDBCListeners

Property name: `kodo.jdbc.JDBCListeners`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getJDBCListeners`

Resource adaptor config-property: `JDBCListeners`

Default: `-`

Description: A comma-separated list of plugin strings (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing `com.solarmetric.jdbc.JDBCListener` event listeners to install. These listeners will be notified on various JDBC-related events. The `com.solarmetric.jdbc.PerformanceTracker` is one such listener that can be used to track JDBC performance.

2.8.6. kodo.jdbc.LRSSize

Property name: `kodo.jdbc.LRSSize`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getLRSSize`

Resource adaptor config-property: `LRSSize`

Default: `query`

Possible values: `query`, `last`, `unknown`

Description: The strategy to use to calculate the size of a result list. This property can also be varied at runtime. See [Section 4.10, “Large Result Sets” \[440\]](#) for details.

2.8.7. kodo.jdbc.MappingDefaults

Property name: `kodo.jdbc.MappingDefaults`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getMappingDefaults`

Resource adaptor config-property: `MappingDefaults`

Default: `default`

Possible values: `default`, `jpa`, `kodo3`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the

`kodo.jdbc.meta.MappingDefaults` to use to define default column names, table names, and constraints for your persistent classes. See [Section 7.5, “Mapping Factory” \[503\]](#) for details.

2.8.8. kodo.jdbc.MappingFactory

Property name: `kodo.jdbc.MappingFactory`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getMappingFactory`

Resource adaptor config-property: `MappingFactory`

Default: -

Possible values: `jdo-orm`, `jdo-table`, `jpa`, `others`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.meta.MetadataFactory` to use to store and retrieve object-relational mapping information for your persistent classes. See [Section 7.5, “Mapping Factory” \[503\]](#) for details.

2.8.9. kodo.jdbc.ResultSetType

Property name: `kodo.jdbc.ResultSetType`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getResultSetType`

Resource adaptor config-property: `ResultSetType`

Default: `forward-only`

Possible values: `forward-only`, `scroll-sensitive`, `scroll-insensitive`

Description: The JDBC result set type to use when fetching result lists. This property can also be varied at runtime. See [Section 4.10, “Large Result Sets” \[440\]](#) for details.

2.8.10. kodo.jdbc.Schema

Property name: `kodo.jdbc.Schema`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getSchema`

Resource adaptor config-property: `Schema`

Default: -

Description: The default schema name to prepend to unqualified table names. Also, the schema in which Kodo will create new tables. See [Section 4.11, “Default Schema” \[443\]](#) for details.

2.8.11. kodo.jdbc.SchemaFactory

Property name: `kodo.jdbc.SchemaFactory`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getSchemaFactory`

Resource adaptor config-property: `SchemaFactory`

Default: `dynamic`

Possible values: `dynamic`, `native`, `file`, `table`, `others`

Description: A plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `kodo.jdbc.schema.SchemaFactory` to use to store and retrieve information about the database schema. See [Section 4.12.2, “Schema Factory” \[443\]](#) for details.

2.8.12. kodo.jdbc.Schemas

Property name: `kodo.jdbc.Schemas`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getSchemas`

Resource adaptor config-property: `Schemas`

Default: -

Description: A comma-separated list of the schemas and/or tables used for your persistent data. See [Section 4.12.1, “Schemas List” \[443\]](#) for details.

2.8.13. kodo.jdbc.SubclassFetchMode

Property name: `kodo.jdbc.SubclassFetchMode`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getSubclassFetchMode`

Resource adaptor config-property: `SubclassFetchMode`

Default: `parallel`

Possible values: `parallel`, `join`, `none`

Description: How to select subclass data when it is in other tables. This setting can also be varied at runtime. See [Section 5.7, “Eager Fetching” \[473\]](#)

2.8.14. kodo.jdbc.SynchronizeMappings

Property name: `kodo.jdbc.SynchronizeMappings`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getSynchronizeMappings`

Resource adaptor config-property: `SynchronizeMappings`

Default: -

Description: Controls whether Kodo will attempt to run the mapping tool on all persistent classes to synchronize their mappings and schema at runtime. Useful for rapid test/debug cycles. See [Section 7.1.4, “Runtime Forward Mapping” \[495\]](#) for more information.

2.8.15. kodo.jdbc.TransactionIsolation

Property name: `kodo.jdbc.TransactionIsolation`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getTransactionIsolation`

Resource adaptor config-property: `TransactionIsolation`

Default: `default`

Possible values: `default`, `none`, `read-committed`, `read-uncommitted`, `repeatable-read`, `serializable`

Description: The JDBC transaction isolation level to use. See [Section 4.5, “Setting the Transaction Isolation” \[436\]](#) for details.

2.8.16. kodo.jdbc.UpdateManager

Property name: `kodo.jdbc.UpdateManager`

Configuration API: `kodo.jdbc.conf.JDBCConfiguration.getUpdateManager`

Resource adaptor config-property: `UpdateManager`

Default: `default`

Description: The full class name of the `kodo.jdbc.kernel.UpdateManager` to use to flush persistent object changes to the datastore. The provided default implementation (`kodo.jdbc.kernel.UpdateManagerImpl`, with a configuration alias of `default`) will suit all but the most advanced users.

Chapter 3. Logging

Logging is an important means of gaining insight into your application's runtime behavior. Kodo provides a flexible logging system that integrates with many existing runtime systems, such as application servers and servlet runners.

There are four built-in logging plugins: a **default logging framework** that covers most needs, a **Log4J** delegate, an **Apache Commons Logging** delegate, and a **no-op** implementation for disabling logging.

Warning

Logging can have a negative impact on performance. Disable verbose logging (such as logging of SQL statements) before running any performance tests. It is advisable to limit or disable logging for a production system. You can disable logging altogether by setting the `kodo.Log` property to `none`.

3.1. Logging Channels

Logging is done over a number of *logging channels*, each of which has a *logging level* which controls the verbosity of log messages recorded for the channel. Kodo uses the following logging channels:

- `kodo.Tool`: Messages issued by the Kodo command line and Ant tools. Most messages are basic statements detailing which classes or files the tools are running on. Detailed output is only available via the logging category the tool belongs to, such as `kodo.Enhance` for the enhancer (see [Section 5.2, “Enhancement” \[452\]](#) or `kodo.MetaData` for the mapping tool (see [Section 7.1, “Forward Mapping” \[489\]](#)) This logging category is provided so that you can get a general idea of what a tool is doing without having to manipulate logging settings that might also affect runtime behavior.
- `kodo.Configuration`: Messages issued by the configuration framework.
- `kodo.Enhance`: Messages pertaining to enhancement and runtime class generation.
- `kodo.MetaData`: Details about the generation of metadata and object-relational mappings.
- `kodo.Runtime`: General Kodo runtime messages.
- `kodo.Query`: Messages about queries. Query strings and any parameter values, if applicable, will be logged to the `TRACE` level at execution time. Information about possible performance concerns will be logged to the `INFO` level.
- `kodo.Remote`: Remote connection and execution messages.
- `kodo.DataCache`: Messages from the L2 data cache plugins.
- `kodo.jdbc.JDBC`: JDBC connection information. General JDBC information will be logged to the `TRACE` level. Information about possible performance concerns will be logged to the `INFO` level.
- `kodo.jdbc.SQL`: This is the most common logging channel to use. Detailed information about the execution of SQL statements will be sent to the `TRACE` level. It is useful to enable this channel if you are curious about the exact SQL that Kodo issues to the datastore.

When using the built-in Kodo logging facilities, you can enable SQL logging by adding `SQL=TRACE` to your `kodo.Log` property.

Kodo can optionally reformat the logged SQL to make it easier to read. To enable pretty-printing, add `PrettyPrint=true` to the `kodo.ConnectionFactoryProperties` property. You can control how many columns wide the pretty-printed SQL will be with the `PrettyPrintLineLength` property. The default line length is 60 columns.

While pretty printing makes things easier to read, it can make output harder to process with tools like `grep`.

Pretty-printing properties configuration might look like so:

JPA XML format:

```
<property name="kodo.Log" value="SQL=TRACE"/>
<property name="kodo.ConnectionFactoryProperties"
  value="MaxActive=100, PrettyPrint=true, PrettyPrintLineLength=72"/>
```

JDO properties format:

```
kodo.Log: SQL=TRACE
kodo.ConnectionFactoryProperties: MaxActive=100, PrettyPrint=true, PrettyPrintLineLength=72
```

- `kodo.jdbc.Schema`: Details about operations on the database schema.
- `kodo.Manage`: JMX and management-related logging channel.
- `kodo.Profile`: Information related to Kodo's profiling framework.

3.2. Kodo Logging

By default, Kodo uses a basic logging framework with the following output format:

```
millis level [thread name] channel - message
```

For example, when loading an application that uses Kodo, a message like the following will be sent to the `kodo.Runtime` channel:

```
2107 INFO [main] kodo.Runtime - Starting Kodo 4.0.0
```

The default logging system accepts the following parameters:

- `File`: The name of the file to log to, or `stdout` or `stderr` to send messages to standard out and standard error, respectively. By default, Kodo sends log messages to standard error.
- `DefaultLevel`: The default logging level of unconfigured channels. Recognized values are `TRACE`, `DEBUG`, `INFO`, `WARN`, and `ERROR`. Defaults to `INFO`.
- `DiagnosticContext`: A string that will be prepended to all log messages.
- `<channel>`: Using the last token of the **logging channel** name, you can configure the log level to use for that channel. See the examples below.

Example 3.1. Standard Kodo Log Configuration

JPA XML format:

```
<property name="kodo.Log" value="DefaultLevel=WARN, Runtime=INFO, Tool=INFO"/>
```

JDO properties format:

```
kodo.Log: DefaultLevel=WARN, Runtime=INFO, Tool=INFO
```

Example 3.2. Standard Kodo Log Configuration + All SQL Statements

JPA XML format:

```
<property name="kodo.Log" value="DefaultLevel=WARN, Runtime=INFO, Tool=INFO, SQL=TRACE"/>
```

JDO properties format:

```
kodo.Log: DefaultLevel=WARN, Runtime=INFO, Tool=INFO, SQL=TRACE
```

Example 3.3. Logging to a File

JPA XML format:

```
<property name="kodo.Log" value="File=/tmp/kodo.log, DefaultLevel=WARN, Runtime=INFO, Tool=INFO"/>
```

JDO properties format:

```
kodo.Log: File=/tmp/kodo.log, DefaultLevel=WARN, Runtime=INFO, Tool=INFO
```

3.3. Disabling Logging

Disabling logging can be useful to analyze performance without any I/O overhead or to reduce verbosity at the console. To do

this, set the `kodo.Log` property to `none`.

Disabling logging permanently, however, will cause all warnings to be consumed. We recommend using one of the more sophisticated mechanisms described in this chapter.

3.4. Log4J

When `kodo.Log` is set to `log4j`, Kodo will delegate to Log4J for logging. In a standalone application, Log4J logging levels are controlled by a resource named `log4j.properties`, which should be available as a top-level resource (either at the top level of a jar file, or in the root of one of the `CLASSPATH` directories). When deploying to a web or EJB application server, Log4J configuration is often performed in a `log4j.xml` file instead of a properties file. For further details on configuring Log4J, please see the [Log4J Manual](#). We present an example `log4j.properties` file below.

Example 3.4. Standard Log4J Logging

```
log4j.rootCategory=WARN, console
log4j.category.kodo.Tool=INFO
log4j.category.kodo.Runtime=INFO
log4j.category.kodo.Remote=WARN
log4j.category.kodo.DataCache=WARN
log4j.category.kodo.MetaData=WARN
log4j.category.kodo.Enhance=WARN
log4j.category.kodo.Query=WARN
log4j.category.kodo.jdbc.SQL=WARN
log4j.category.kodo.jdbc.JDBC=WARN
log4j.category.kodo.jdbc.Schema=WARN

log4j.appender.console=org.apache.log4j.ConsoleAppender
```

3.5. Apache Commons Logging

Set the `kodo.Log` property to `commons` to use the [Apache Jakarta Commons Logging](#) thin library for issuing log messages. The Commons Logging libraries act as a wrapper around a number of popular logging APIs, including the [Jakarta Log4J](#) project, and the native [java.util.logging](#) package in JDK 1.4. If neither of these libraries are available, then logging will fall back to using simple console logging.

When using the Commons Logging framework in conjunction with Log4J, configuration will be the same as was discussed in the Log4J section above.

3.5.1. JDK 1.4 java.util.logging

When using JDK 1.4 or higher in conjunction with Kodo's Commons Logging support, logging will proceed through Java's built-in logging provided by the [java.util.logging](#) package. For details on configuring the built-in logging system, please see the [Java Logging Overview](#).

By default, JDK 1.4's logging package looks in the `JAVA_HOME/lib/logging.properties` file for logging configuration. This can be overridden with the `java.util.logging.config.file` system property. For example:

```
java -Djava.util.logging.config.file=mylogging.properties com.company.MyClass
```

Example 3.5. JDK 1.4 Log Properties

```

# specify the handlers to create in the root logger
# (all loggers are children of the root logger)
# the following creates two handlers
handlers=java.util.logging.ConsoleHandler, java.util.logging.FileHandler

# set the default logging level for the root logger
.level=ALL

# set the default logging level for new ConsoleHandler instances
java.util.logging.ConsoleHandler.level=INFO

# set the default logging level for new FileHandler instances
java.util.logging.FileHandler.level=ALL

# set the default formatter for new ConsoleHandler instances
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

# set the default logging level for all Kodo logs
kodo.Tool.level=INFO
kodo.Runtime.level=INFO
kodo.Remote.level=INFO
kodo.DataCache.level=INFO
kodo.Metadata.level=INFO
kodo.Enhance.level=INFO
kodo.Query.level=INFO
kodo.jdbc.SQL.level=INFO
kodo.jdbc.JDBC.level=INFO
kodo.jdbc.Schema.level=INFO

```

3.6. Custom Log

If none of available logging systems meet your needs, you can configure the logging system with a custom logger. You might use custom logging to integrate with a proprietary logging framework used by some applications servers, or for logging to a graphical component for GUI applications.

A custom logging framework must include an implementation of the `com.solarmetric.log.LogFactory` interface. We present a custom `LogFactory` below.

Example 3.6. Custom Logging Class

```

package com.xyz;

import com.solarmetric.log.*;

public class CustomLogFactory
    implements LogFactory
{
    private String _prefix = "CUSTOM LOG";

    public void setPrefix (String prefix)
    {
        _prefix = prefix;
    }

    public Log getLog (String channel)
    {
        // Return a simple extension of AbstractLog that will log
        // everything to the System.err stream. Note that this is
        // roughly equivalent to Kodo's default logging behavior.
        return new AbstractLog ()
        {
            protected boolean isEnabled (short logLevel)
            {
                // log all levels
                return true;
            }
        }
    }
}

```

```
protected void log (short type, String message, Throwable t)
{
    // just send everything to System.err
    System.err.println (_prefix + ": " + type + ": "
        + message + ": " + t);
};
}
```

To make Kodo use your custom log factory, set the `kodo.Log` configuration property to your factory's full class name. Because this property is a plugin property (see [Section 2.5, “Plugin Configuration” \[395\]](#)), you can also pass parameters to your factory. For example, to use the example factory above and set its prefix to "LOG MSG", you would set the `kodo.Log` property to the following string:

```
com.xyz.CustomLogFactory(Prefix="LOG MSG")
```

Chapter 4. JDBC

Kodo JPA/JDO uses a relational database for object persistence. It communicates with the database using the Java DataBase Connectivity (JDBC) APIs. This chapter describes how to configure Kodo to work with the JDBC driver for your database, and how to access JDBC functionality at runtime.

4.1. Using the Kodo DataSource

Kodo includes its own `javax.sql.DataSource` implementation, complete with configurable connection pooling and prepared statement caching. If you choose to use Kodo's `DataSource`, then you must specify the following properties:

Note

JDO users can specify the equivalent JDO properties rather than the Kodo properties below.

- `kodo.ConnectionUserName`: The JDBC user name for connecting to the database.
- `kodo.ConnectionPassword`: The JDBC password for the above user.
- `kodo.ConnectionURL`: The JDBC URL for the database.
- `kodo.ConnectionDriverName`: The JDBC driver class.

To configure advanced features such as connection pooling and prepared statement caching, or to configure the underlying JDBC driver, use the following optional properties. The syntax of these property strings follows the syntax of Kodo plugin parameters described in [Section 2.5, “Plugin Configuration” \[395\]](#).

- **`kodo.ConnectionProperties`**: If the listed driver is an instance of `java.sql.Driver`, this string will be parsed into a `Properties` instance, which will then be used to obtain database connections through the `Driver.connect(String url, Properties props)` method. If, on the other hand, the listed driver is a `javax.sql.DataSource`, the string will be treated as a plugin properties string, and matched to the bean setter methods of the `DataSource` instance.
- **`kodo.ConnectionFactoryProperties`**: Kodo's built-in `DataSource` allows you to set the following options via this plugin string:
 - **`ExceptionAction`**: The action to take when when a connection that has thrown an exception is returned to the pool. Set to `destroy` to destroy the connection. Set to `validate` to validate the connection (subject to the `TestOnReturn`, `TestOnBorrow`, and other test settings). Set to `none` to ignore the fact that the connection has thrown an exception, and assume it is still usable. Defaults to `destroy`.
 - **`MaxActive`**: The maximum number of database connections in use at one time. A value of 0 disables connection pooling. Defaults to 8. This is the maximum number of connections that Kodo will give out to your application. If a connection is requested while `MaxActive` other connections are in use, Kodo will wait for `MaxWait` milliseconds for a connection to be returned, and will then throw an exception if no connection was made available.
 - **`MaxIdle`**: The maximum number of idle database connections to keep in the pool. Defaults to 8. If this number is less than `MaxActive`, then Kodo will close extra connections that are returned to the pool if there are already `MaxIdle` available connections. This allows for unexpected or atypical load while still maintaining a relatively small pool when there is less load on the system.
 - **`MaxTotal`**: The maximum number of database connections in the pool, whether active or idle. Defaults to -1, meaning no limit (the limit will be dictated by `MaxActive` and `MaxIdle` for each unique user name).

- **MaxWait:** The maximum number of milliseconds to wait for a free database connection to become available before giving up. Defaults to 3000.
- **MinEvictableIdleTimeMillis:** The minimum number of milliseconds that a database connection can sit idle before it becomes a candidate for eviction from the pool. Defaults to 30 minutes. Set to 0 to never evict a connection based on idle time alone.
- **RollbackOnReturn:** Force all connections to be rolled back when they are returned to the pool. If false, the `DataSource` will only roll back connections when it detects that there have been any transactional updates on the connection.
- **TestOnBorrow:** Whether to validate database connections before obtaining them from the pool. Note that validation only consists of a call to the connection's `isClosed` method unless you specify a `ValidationSQL` string. Defaults to `true`.
- **TestOnReturn:** Set to `true` to validate database connections when they are returned to the pool. Note that validation only consists of a call to the connection's `isClosed` method unless you specify a `ValidationSQL` string.
- **TestWhileIdle:** Set to `true` to periodically validate idle database connections.
- **TimeBetweenEvictionRunsMillis:** The number of milliseconds between runs of the eviction thread. Defaults to `-1`, meaning the eviction thread will never run.
- **TrackParameters:** When `true`, Kodo will track the parameters that were set for all `PreparedStatement`s that are executed or batched so that they can be included in error messages. Defaults to `true`.
- **ValidationSQL:** A simple SQL query to issue to validate a database connection. If this property is not set, then the only validation performed is to use the `Connection.isClosed` method. The following table shows the default settings for different databases. If a database is not shown, this property defaults to null.

Table 4.1. Validation SQL Defaults

Database	SQL
DB2	SELECT DISTINCT(CURRENT TIMESTAMP) FROM SYS-IBM.SYSTABLES
Empress	SELECT DISTINCT(TODAY) FROM SYS_TABLES
Informix	SELECT DISTINCT CURRENT TIMESTAMP FROM INFORMIX.SYSTABLES
MySQL	SELECT NOW()
Oracle	SELECT SYSDATE FROM DUAL
Postgres	SELECT NOW()
SQLServer	SELECT GETDATE()
Sybase	SELECT GETDATE()

To disable validation SQL, set this property to an empty string, as in **Example 4.1, “Properties for the Kodo DataSource” [425]**

- **ClosePoolSQL:** A simple SQL statement to execute when the connection pool is completely closed. This can be used, for example, to cleanly issue a shutdown statement to a file-based database.
- **ValidationTimeout:** The minimum number of milliseconds that must elapse before a connection will ever be re-validated. This property is typically used with `TestOnBorrow` or `TestOnReturn` to reduce the number of validations performed, because the same connection is often borrowed and returned many times in a short span. Defaults to 300000 (5 minutes).

- **WarningAction:** The action to take when a `SQLWarning` is detected on a connection. Possible values are:
 - `ignore`: Warnings will not be checked for, and will be ignored. This is the default.
 - `trace`: The warning will be logged on the `TRACE` channel of the JDBC log.
 - `info`: The warning will be logged on the `INFO` channel of the JDBC log.
 - `warn`: The warning will be logged on the `WARN` channel of the JDBC log.
 - `error`: The warning will be logged on the `ERROR` channel of the JDBC log.
 - `throw`: All `SQLWarning` instances will be thrown as if they were errors.
 - `handle`: The `SQLWarning` instance will be passed through the `handleWarning` method of `kodo.jdbc.sql.DBDictionary`, which allows a custom extension of the dictionary to use heuristic-based warning handling.
- **WhenExhaustedAction:** The action to take when there are no available database connections in the pool. Set to `exception` to immediately throw an exception. Set to `block` to block until a connection is available or the maximum wait time is exceeded. Set to `grow` to automatically grow the pool. Defaults to `block`.

Additionally, the following properties are available whether you use Kodo's built-in `DataSource` or a third-party's:

- **MaxCachedStatements:** The maximum number of `java.sql.PreparedStatements` to cache. Statement caching can dramatically speed up some databases. Defaults to 50 for Kodo's `DataSource`, and 0 for third-party `DataSources`. Most third-party `DataSources` do not benefit from Kodo's prepared statement cache, because each returned connection has a unique hash code, making it impossible for Kodo to match connections to their cached statements.
- **QueryTimeout:** The maximum number of seconds the JDBC driver will wait for a statement to execute.

Example 4.1. Properties for the Kodo DataSource

JPA XML format:

```
<property name="kodo.ConnectionUserName" value="user"/>
<property name="kodo.ConnectionPassword" value="pass"/>
<property name="kodo.ConnectionURL" value="jdbc:hsqldb:db-hypersonic"/>
<property name="kodo.ConnectionDriverName" value="org.hsqldb.jdbcDriver"/>
<property name="kodo.ConnectionFactoryProperties"
  value="MaxActive=50, MaxIdle=10, ValidationTimeout=50000, MaxCachedStatements=100, ValidationSQL='"/>
```

JDO properties format:

```
kodo.ConnectionUserName: user
kodo.ConnectionPassword: pass
kodo.ConnectionURL: jdbc:hsqldb:db-hypersonic
kodo.ConnectionDriverName: org.hsqldb.jdbcDriver
kodo.ConnectionFactoryProperties: MaxActive=50, MaxIdle=10, \
  ValidationTimeout=50000, MaxCachedStatements=100, ValidationSQL=''
```

4.2. Using a Third-Party DataSource

You can use Kodo with any third-party `javax.sql.DataSource`. There are multiple ways of telling Kodo about a DataSource:

- Set the DataSource into the map passed to `Persistence.createEntityManagerFactory` under the `kodo.ConnectionFactory` key.
- Bind the DataSource into JNDI, and then specify its location in the `jta-data-source` or `non-jta-data-source` element of the **JPA XML format** (depending on whether the DataSource is managed by JTA), or in the `kodo.ConnectionFactoryName` property.
- Specify the full class name of the DataSource implementation in the `kodo.ConnectionDriverName` property in place of a JDBC driver. In this configuration Kodo will instantiate an instance of the named class via reflection. It will then configure the DataSource with the properties in the `kodo.ConnectionProperties` setting.

Some advanced features of Kodo's own DataSource can also be used with third-party implementations. Kodo layers on top of the third-party DataSource to provide the extra functionality. To configure these advanced features, including prepared statement caching, use the `kodo.ConnectionFactoryProperties` property described in the previous section.

Example 4.2. Properties File for a Third-Party DataSource

JPA XML format:

```
<property name="kodo.ConnectionDriverName" value="oracle.jdbc.pool.OracleDataSource" />
<property name="kodo.ConnectionProperties"
  value="PortNumber=1521, ServerName=saturn, DatabaseName=solarsid, DriverType=thin" />
<property name="kodo.ConnectionFactoryProperties" value="QueryTimeout=5000" />
```

JDO properties format:

```
kodo.ConnectionDriverName: oracle.jdbc.pool.OracleDataSource
kodo.ConnectionProperties: PortNumber=1521, ServerName=saturn, \
  DatabaseName=solarsid, DriverType=thin
kodo.ConnectionFactoryProperties: QueryTimeout=5000
```

4.2.1. Managed and XA DataSources

Certain application servers automatically enlist their DataSources in global transactions. When this is the case, Kodo should not attempt to commit the underlying connection, leaving JDBC transaction completion to the application server. To notify Kodo that your third-party DataSource is managed by the application server, set the `kodo.ConnectionFactoryMode` property to `managed`.

Note that Kodo can only use managed DataSources when it is also integrating with the application server's managed transactions, as discussed in [Section 8.2, “Integrating with the Transaction Manager”](#) [535]. Also note that all XA DataSources are enlisted, and you must set this property when using any XA DataSource. XA transactions are detailed in [Section 8.3, “XA Transactions”](#) [536].

When using a managed DataSource, you should also configure a second unmanaged DataSource that Kodo can use to per-

form tasks that are independent of the global transaction. The most common of these tasks is updating the sequence table Kodo uses to generate unique primary key values for your datastore identity objects. Configure the second `DataSource` just as the first, but use the various "2" connection properties, such as `kodo.ConnectionFactory2Name` or `kodo.Connection2DriverName`. These properties are outlined in [Chapter 2, Configuration \[393\]](#). If your second `DataSource` is also bound to JNDI, you can use JPA XML's `non-jta-data-source` to specify its location. Typically, though, you will use Kodo's built-in implementation for the second `DataSource` (see [Section 4.1, "Using the Kodo DataSource" \[423\]](#)).

Example 4.3. Managed DataSource Configuration

JPA XML format:

```
<!-- managed DataSource -->
<jta-data-source>java:/OracleXASource</jta-data-source>
<properties>
  <!-- use Kodo's built-in DataSource for unmanaged connections -->
  <property name="kodo.Connection2UserName" value="scott"/>
  <property name="kodo.Connection2Password" value="tiger"/>
  <property name="kodo.Connection2URL" value="jdbc:oracle:thin:@CROM:1521:KodoDB"/>
  <property name="kodo.Connection2DriverName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="kodo.ConnectionFactory2Properties" value="MaxActive=20, MaxIdle=10"/>
  <!-- managed transaction and enlisted configuration -->
  <property name="kodo.TransactionMode" value="managed"/>
  <property name="kodo.ConnectionFactoryMode" value="managed"/>
</properties>
```

JDO properties format:

```
# managed DataSource
kodo.ConnectionFactoryName: java:/OracleXASource

# use Kodo's built-in DataSource for unmanaged connections
kodo.Connection2UserName: scott
kodo.Connection2Password: tiger
kodo.Connection2URL: jdbc:oracle:thin:@CROM:1521:KodoDB
kodo.Connection2DriverName: oracle.jdbc.driver.OracleDriver
kodo.ConnectionFactory2Properties: MaxActive=20, MaxIdle=10

# managed transaction and enlisted configuration
kodo.TransactionMode: managed
kodo.ConnectionFactoryMode: managed
```

4.3. Runtime Access to DataSource

The JPA standard defines how to access JDBC connections from enterprise beans. Kodo also provides APIs to access an `EntityManager`'s connection, or to retrieve a connection directly from the `EntityManagerFactory`'s `DataSource`.

The `KodoEntityManager.getConnection` method returns an `EntityManager`'s connection. If the `EntityManager` does not already have a connection, it will obtain one. The returned connection is only guaranteed to be transactionally consistent with other `EntityManager` operations if the `EntityManager` is in a managed or non-optimistic transaction, if the `EntityManager` has flushed in the current transaction, or if you have used the `KodoEntityManager.beginStore` method to ensure that a datastore transaction is in progress. Always close the returned connection before attempting any other `EntityManager` operations. Kodo will ensure that the underlying native connection is not released if a datastore transaction is in progress.

Example 4.4. Using the EntityManager's Connection

```

import java.sql.*;
import kodo.persistence.*;

...

KodoEntityManager kem = KodoPersistence.cast (em);
Connection conn = (Connection) kem.getConnection ();

// do JDBC stuff

conn.close ();

```

The example below shows how to use a connection directly from the `DataSource`, rather than using an `EntityManager`'s connection.

Example 4.5. Using the EntityManagerFactory's DataSource

```

import java.sql.*;
import javax.sql.*;
import kodo.conf.*;
import kodo.persistence.*;

...

KodoEntityManagerFactory kemf = KodoPersistence.cast (emf);
KodoConfiguration conf = kemf.getConfiguration ();
DataSource dataSource = (DataSource) conf.getConnectionFactory ();
Connection conn = dataSource.getConnection ();

// do JDBC stuff

conn.close ();

```

Section 8.13, “Connection Access” [224] of the JDO Overview shows how to access a `PersistenceManager`'s JDBC connection. Kodo also allows you to get connections directly from the `PersistenceManagerFactory`'s `DataSource`.

Example 4.6. Using the PersistenceManagerFactory DataSource

```

import java.sql.*;
import javax.sql.*;
import kodo.conf.*;
import kodo.jdo.*;

...

KodoPersistenceManagerFactory kpmf = KodoJDOHelper.cast (pmf);
KodoConfiguration conf = kpmf.getConfiguration ();
DataSource dataSource = (DataSource) conf.getConnectionFactory ();
Connection conn = dataSource.getConnection ();

// do JDBC stuff

conn.close ();

```

4.4. Database Support

Kodo JPA/JDO can take advantage of any JDBC 2.x compliant driver, making almost any major database a candidate for use. See our officially supported database list in [Appendix 4, Supported Databases \[629\]](#) for more information. Typically, Kodo auto-configures its JDBC behavior and SQL dialect for your database, based on the values of your connection-related configuration properties.

If Kodo cannot detect what type of database you are using, or if you are using an unsupported database, you will have to tell Kodo what `kodo.jdbc.sql.DBDictionary` to use. The `DBDictionary` abstracts away the differences between databases. You can plug a dictionary into Kodo using the `kodo.jdbc.DBDictionary` configuration property. The built-in dictionaries are listed below. If you are using an unsupported database, you may have to write your own `DBDictionary` subclass, a simple process.

- `access`: Dictionary for Microsoft Access. This is an alias for the `kodo.jdbc.sql.AccessDictionary` class.
- `db2`: Dictionary for IBM's DB2 database. This is an alias for the `kodo.jdbc.sql.DB2Dictionary` class.
- `derby`: Dictionary for the Apache Derby database. This is an alias for the `kodo.jdbc.sql.DerbyDictionary` class.
- `empres`: Dictionary for Empress database This is an alias for the `kodo.jdbc.sql.EmpressDictionary` class.
- `foxpro`: Dictionary for Microsoft Visual FoxPro. This is an alias for the `kodo.jdbc.sql.FoxProDictionary` class.
- `hsqldb`: Dictionary for the Hypersonic SQL database. This is an alias for the `kodo.jdbc.sql.HSQLDictionary` class.
- `informix`: Dictionary for the Informix database. This is an alias for the `kodo.jdbc.sql.InformixDictionary` class.
- `jdatastore`: Dictionary for Borland JDataStore. This is an alias for the `kodo.jdbc.sql.JDataStoreDictionary` class.
- `mysql`: Dictionary for the MySQL database. This is an alias for the `kodo.jdbc.sql.MySQLDictionary` class.
- `oracle`: Dictionary for Oracle. This is an alias for the `kodo.jdbc.sql.OracleDictionary` class.
- `pointbase`: Dictionary for Pointbase Embedded database. This is an alias for the `kodo.jdbc.sql.PointbaseDictionary` class.
- `postgres`: Dictionary for PostgreSQL. This is an alias for the `kodo.jdbc.sql.PostgresDictionary` class.
- `sqlserver`: Dictionary for Microsoft's SQLServer database. This is an alias for the `kodo.jdbc.sql.SQLServerDictionary` class.
- `sybase`: Dictionary for Sybase. This is an alias for the `kodo.jdbc.sql.SybaseDictionary` class.

The example below demonstrates how to set a dictionary and configure its properties in your configuration file. The `DBDictionary` property uses Kodo's [plugin syntax](#).

Example 4.7. Specifying a DBDictionary

JPA XML format:

```
<property name="kodo.jdbc.DBDictionary" value="hsqldb(SimulateLocking=true)" />
```

JDO properties format:

```
kodo.jdbc.DBDictionary: hsql(SimulateLocking=true)
```

4.4.1. DBDictionary Properties

The standard dictionaries all recognize the following properties. These properties will usually not need to be overridden, since the dictionary implementation should use the appropriate default values for your database. You typically won't use these properties unless you are designing your own `DBDictionary` for an unsupported database.

- `DriverVendor`: The vendor of the particular JDBC driver you are using. Some dictionaries must alter their behavior depending on the driver vendor. See the `VENDOR_XXX` constants defined in your dictionary's Javadoc for available options.
- `CatalogSeparator`: The string the database uses to delimit between the schema name and the table name. This is typically `" . "`, which is the default.
- `CreatePrimaryKeys`: If `false`, then do not create database primary keys for identifiers. Defaults to `true`.
- `ConstraintNameMode`: When creating constraints, whether to put the constraint name before the definition (`before`), just after the constraint type name (`mid`), or after the constraint definition (`after`). Defaults to `before`.
- `MaxTableNameLength`: The maximum number of characters in a table name. Defaults to 128.
- `MaxColumnNameLength`: The maximum number of characters in a column name. Defaults to 128.
- `MaxConstraintNameLength`: The maximum number of characters in a constraint name. Defaults to 128.
- `MaxIndexNameLength`: The maximum number of characters in an index name. Defaults to 128.
- `MaxAutoAssignNameLength`: Set this property to the maximum length of name for sequences used for auto-increment columns. Names longer than this value are truncated. Defaults to 31.
- `MaxIndexesPerTable`: The maximum number of indexes that can be placed on a single table. Defaults to no limit.
- `SupportsForeignKeys`: Whether the database supports foreign keys. Defaults to `true`.
- `SupportsUniqueConstraints`: Whether the database supports unique constraints. Defaults to `true`.
- `SupportsDeferredConstraints`: Whether the database supports deferred constraints. Defaults to `true`.
- `SupportsRestrictDeleteAction`: Whether the database supports the `RESTRICT` foreign key delete action. Defaults to `true`.
- `SupportsCascadeDeleteAction`: Whether the database supports the `CASCADE` foreign key delete action. Defaults to `true`.
- `SupportsNullDeleteAction`: Whether the database supports the `SET NULL` foreign key delete action. Defaults to `true`.
- `SupportsDefaultDeleteAction`: Whether the database supports the `SET DEFAULT` foreign key delete action. Defaults to `true`.
- `SupportsAlterTableWithAddColumn`: Whether the database supports adding a new column in an `ALTER TABLE` statement. Defaults to `true`.
- `SupportsAlterTableWithDropColumn`: Whether the database supports dropping a column in an `ALTER TABLE`

statement. Defaults to `true`.

- `ReservedWords`: A comma-separated list of reserved words for this database, beyond the standard SQL92 keywords.
- `SystemTables`: A comma-separated list of table names that should be ignored.
- `SystemSchemas`: A comma-separated list of schema names that should be ignored.
- `SchemaCase`: The case to use when querying the database metadata about schema components. Defaults to making all names upper case. Available values are: `upper`, `lower`, `preserve`.
- `ValidationSQL`: The SQL used to validate that a connection is still in a valid state. For example, "SELECT SYSDATE FROM DUAL" for Oracle.
- `InitializationSQL`: A piece of SQL to issue against the database whenever a connection is retrieved from the Data-Source.
- `JoinSyntax`: The SQL join syntax to use in select statements. See **Section 4.6, "Setting the SQL Join Syntax" [436]**
- `CrossJoinClause`: The clause to use for a cross join (cartesian product). Defaults to `CROSS JOIN`.
- `InnerJoinClause`: The clause to use for an inner join. Defaults to `INNER JOIN`.
- `OuterJoinClause`: The clause to use for an left outer join. Defaults to `LEFT OUTER JOIN`.
- `RequiresConditionForCrossJoin`: Some databases require that there always be a conditional statement for a cross join. If set, this parameter ensures that there will always be some condition to the join clause.
- `ToUpperCaseFunction`: SQL function call for for converting a string to upper case. Use the token `{0}` to represent the argument.
- `ToLowerCaseFunction`: Name of the SQL function for converting a string to lower case. Use the token `{0}` to represent the argument.
- `StringLengthFunction`: Name of the SQL function for getting the length of a string. Use the token `{0}` to represent the argument.
- `SubstringFunctionName`: Name of the SQL function for getting the substring of a string.
- `DistinctCountColumnSeparator`: The string the database uses to delimit between column expressions in a `SELECT COUNT(DISTINCT column-list)` clause. Defaults to null for most databases, meaning that multiple columns in a distinct `COUNT` clause are not supported.
- `ForUpdateClause`: The clause to append to `SELECT` statements to issue queries that obtain pessimistic locks. Defaults to `FOR UPDATE`.
- `TableForUpdateClause`: The clause to append to the end of each table alias in queries that obtain pessimistic locks. Defaults to null.
- `SupportsSelectForUpdate`: If true, then the database supports `SELECT` statements with a pessimistic locking clause. Defaults to true.
- `SupportsLockingWithDistinctClause`: If true, then the database supports `FOR UPDATE` select clauses with `DISTINCT` clauses.
- `SupportsLockingWithOuterJoin`: If true, then the database supports `FOR UPDATE` select clauses with outer join queries.
- `SupportsLockingWithInnerJoin`: If true, then the database supports `FOR UPDATE` select clauses with inner join queries.

- `SupportsLockingWithMultipleTables`: If true, then the database supports FOR UPDATE select clauses that select from multiple tables.
- `SupportsLockingWithOrderClause`: If true, then the database supports FOR UPDATE select clauses with ORDER BY clauses.
- `SupportsLockingWithSelectRange`: If true, then the database supports FOR UPDATE select clauses with queries that select a range of data using LIMIT, TOP or the database equivalent. Defaults to true.
- `SimulateLocking`: Some databases do not support pessimistic locking, which will result in an exception when you attempt a pessimistic transaction. Setting this property to true bypasses the locking check to allow pessimistic transactions even on databases that do not support locking. Defaults to false.
- `SupportsQueryTimeout`: If true, then the JDBC driver supports calls to `java.sql.Statement.setQueryTimeout`.
- `SupportsHaving`: Whether this database supports HAVING clauses in selects.
- `SupportsSelectStartIndex`: Whether this database can create a select that skips the first N results.
- `SupportsSelectEndIndex`: Whether this database can create a select that is limited to the first N results.
- `SupportsSubselect`: Whether this database supports subselects in queries.
- `RequiresAliasForSubselect`: If true, then the database requires that subselects in a FROM clause be assigned an alias.
- `BatchLimit`: The maximum number of SQL update statements to batch together. Set to 0 to disable statement batching, or -1 for no limit. See [Section 4.9, “Statement Batching” \[440\]](#)
- `BatchParameterLimit`: The maximum number of parameters that can be batched together for a single batch update. Some databases can only handle a certain total number of prepared statement parameters in a single batch. This value will cause Kodo to flush a SQL batch once the number of batched statements times the number of bound parameters per statement exceeds this value. Set to 0 to disable SQL batching, or -1 for no limit.
- `SupportsUpdateCountsForBatch`: Whether the JDBC driver correctly returns the set of update counts when a batch statement is executed.
- `SupportsTotalCountsForBatch`: If a JDBC driver doesn't support batch update counts, whether it at least returns the total number of updates made when a batch statement is executed.
- `SupportsMultipleNontransactionalResultSets`: If true, then a nontransactional connection is capable of having multiple open `ResultSet` instances.
- `StorageLimitationsFatal`: If true, then any data truncation/rounding that is performed by the dictionary in order to store a value in the database will be treated as a fatal error, rather than just issuing a warning.
- `StoreLargeNumbersAsStrings`: Many databases have limitations on the number of digits that can be stored in a numeric field (for example, Oracle can only store 38 digits). For applications that operate on very large `BigInteger` and `BigDecimal` values, it may be necessary to store these objects as string fields rather than the database's numeric type. Note that this may prevent meaningful numeric queries from being executed against the database. Defaults to false.
- `StoreCharsAsNumbers`: Set this property to false to store Java char fields as CHAR values rather than numbers. Defaults to true.
- `UseGetBytesForBlobs`: If true, then `ResultSet.getBytes` will be used to obtain blob data rather than `ResultSet.getBinaryStream`.
- `UseGetObjectForBlobs`: If true, then `ResultSet.getObject` will be used to obtain blob data rather than `ResultSet.getBinaryStream`.

- `UseSetBytesForBlobs`: If true, then `PreparedStatement.setBytes` will be used to set blob data, rather than `PreparedStatement.setBinaryStream`.
- `UseGetStringForClobs`: If true, then `ResultSet.getString` will be used to obtain clob data rather than `ResultSet.getCharacterStream`.
- `UseSetStringForClobs`: If true, then `PreparedStatement.setString` will be used to set clob data, rather than `PreparedStatement.setCharacterStream`.
- `CharacterColumnSize`: The default size of `varchar` and `char` columns. Typically 255.
- `ArrayTypeNames`: The overridden default column type for `java.sql.Types.ARRAY`. This is only used when the schema is generated by the `mappingtool`.
- `BigIntTypeName`: The overridden default column type for `java.sql.Types.BIGINT`. This is only used when the schema is generated by the `mappingtool`.
- `BinaryTypeName`: The overridden default column type for `java.sql.Types.BINARY`. This is only used when the schema is generated by the `mappingtool`.
- `BitTypeName`: The overridden default column type for `java.sql.Types.BIT`. This is only used when the schema is generated by the `mappingtool`.
- `BlobTypeName`: The overridden default column type for `java.sql.Types.BLOB`. This is only used when the schema is generated by the `mappingtool`.
- `CharTypeName`: The overridden default column type for `java.sql.Types.CHAR`. This is only used when the schema is generated by the `mappingtool`.
- `ClobTypeName`: The overridden default column type for `java.sql.Types.CLOB`. This is only used when the schema is generated by the `mappingtool`.
- `DateTypeName`: The overridden default column type for `java.sql.Types.DATE`. This is only used when the schema is generated by the `mappingtool`.
- `DecimalTypeName`: The overridden default column type for `java.sql.Types.DECIMAL`. This is only used when the schema is generated by the `mappingtool`.
- `DistinctTypeName`: The overridden default column type for `java.sql.Types.DISTINCT`. This is only used when the schema is generated by the `mappingtool`.
- `DoubleTypeName`: The overridden default column type for `java.sql.Types.DOUBLE`. This is only used when the schema is generated by the `mappingtool`.
- `FloatTypeName`: The overridden default column type for `java.sql.Types.FLOAT`. This is only used when the schema is generated by the `mappingtool`.
- `IntegerTypeName`: The overridden default column type for `java.sql.Types.INTEGER`. This is only used when the schema is generated by the `mappingtool`.
- `JavaObjectTypeNames`: The overridden default column type for `java.sql.Types.JAVAOBJECT`. This is only used when the schema is generated by the `mappingtool`.
- `LongVarbinaryTypeNames`: The overridden default column type for `java.sql.Types.LONGVARBINARY`. This is only used when the schema is generated by the `mappingtool`.
- `LongVarcharTypeNames`: The overridden default column type for `java.sql.Types.LONGVARCHAR`. This is only used when the schema is generated by the `mappingtool`.
- `NullTypeNames`: The overridden default column type for `java.sql.Types.NULL`. This is only used when the schema is

generated by the `mappingtool`.

- `NumericTypeName`: The overridden default column type for `java.sql.Types.NUMERIC`. This is only used when the schema is generated by the `mappingtool`.
- `OtherTypeName`: The overridden default column type for `java.sql.Types.OTHER`. This is only used when the schema is generated by the `mappingtool`.
- `RealTypeName`: The overridden default column type for `java.sql.Types.REAL`. This is only used when the schema is generated by the `mappingtool`.
- `RefTypeName`: The overridden default column type for `java.sql.Types.REF`. This is only used when the schema is generated by the `mappingtool`.
- `SmallintTypeName`: The overridden default column type for `java.sql.Types.SMALLINT`. This is only used when the schema is generated by the `mappingtool`.
- `StructTypeName`: The overridden default column type for `java.sql.Types.STRUCT`. This is only used when the schema is generated by the `mappingtool`.
- `TimeTypeName`: The overridden default column type for `java.sql.Types.TIME`. This is only used when the schema is generated by the `mappingtool`.
- `TimestampTypeName`: The overridden default column type for `java.sql.Types.TIMESTAMP`. This is only used when the schema is generated by the `mappingtool`.
- `TinyintTypeName`: The overridden default column type for `java.sql.Types.TINYINT`. This is only used when the schema is generated by the `mappingtool`.
- `VarbinaryTypeName`: The overridden default column type for `java.sql.Types.VARBINARY`. This is only used when the schema is generated by the `mappingtool`.
- `VarcharTypeName`: The overridden default column type for `java.sql.Types.VARCHAR`. This is only used when the schema is generated by the `mappingtool`.
- `UseSchemaName`: If `false`, then avoid including the schema name in table name references. Defaults to `true`.
- `TableTypes`: Comma-separated list of table types to use when looking for tables during schema reflection, as defined in the `java.sql.DatabaseMetaData.getTableInfo` JDBC method. An example is: `"TABLE,VIEW,ALIAS"`. Defaults to `"TABLE"`.
- `SupportsSchemaForGetTables`: If `false`, then the database driver does not support using the schema name for schema reflection on table names.
- `SupportsSchemaForGetColumns`: If `false`, then the database driver does not support using the schema name for schema reflection on column names.
- `SupportsNullTableForGetColumns`: If `true`, then the database supports passing a `null` parameter to `DatabaseMetaData.getColumns` as an optimization to get information about all the tables. Defaults to `true`.
- `SupportsNullTableForGetPrimaryKeys`: If `true`, then the database supports passing a `null` parameter to `DatabaseMetaData.getPrimaryKeys` as an optimization to get information about all the tables. Defaults to `false`.
- `SupportsNullTableForGetIndexInfo`: If `true`, then the database supports passing a `null` parameter to `DatabaseMetaData.getIndexInfo` as an optimization to get information about all the tables. Defaults to `false`.
- `SupportsNullTableForGetImportedKeys`: If `true`, then the database supports passing a `null` parameter to `DatabaseMetaData.getImportedKeys` as an optimization to get information about all the tables. Defaults to `false`.
- `UseGetBestRowIdentifierForPrimaryKeys`: If `true`, then metadata queries will use `Database-`

`MetaData.getBestRowIdentifier` to obtain information about primary keys, rather than `DatabaseMetaData.getPrimaryKeys`.

- `RequiresAutoCommitForMetadata`: If true, then the JDBC driver requires that autocommit be enabled before any schema interrogation operations can take place.
- `AutoAssignClause`: The column definition clause to append to a creation statement. For example, "AUTO_INCREMENT" for MySQL. This property is set automatically in the dictionary, and should not need to be overridden, and is only used when the schema is generated using the `mappingtool`.
- `AutoAssignTypeName`: The column type name for auto-increment columns. For example, "SERIAL" for PostgreSQL. This property is set automatically in the dictionary, and should not need to be overridden, and is only used when the schema is generated using the `mappingtool`.
- `LastGeneratedKeyQuery`: The query to issue to obtain the last automatically generated key for an auto-increment column. For example, "select @@identity" for Sybase. This property is set automatically in the dictionary, and should not need to be overridden.
- `NextSequenceQuery`: A SQL string for obtaining a native sequence value. May use a placeholder of {0} for the variable sequence name. Defaults to a database-appropriate value.

4.4.2. MySQLDictionary Properties

The `mysql` dictionary also understands the following properties:

- `DriverDeserializesBlobs`: Many MySQL drivers automatically deserialize BLOBs on calls to `ResultSet.getObject`. The `MySQLDictionary` overrides the standard `DBDictionary.getBlobObject` method to take this into account. If your driver does not deserialize automatically, set this property to `false`.
- `TableType`: The MySQL table type to use when creating tables. Defaults to `innodb`.
- `UseClobs`: Some older versions of MySQL do not handle clobs correctly. To enable clob functionality, set this to `true`. Defaults to `false`.

4.4.3. OracleDictionary Properties

The `oracle` dictionary understands the following additional properties:

- `UseTriggersForAutoAssign`: If true, then Kodo will allow simulation of auto-increment columns by the use of Oracle triggers. Kodo will assume that the current sequence value from the sequence specified in the `AutoAssignSequenceName` parameter will hold the value of the new primary key for rows that have been inserted. For more details on auto-increment support, see [Section 5.3.3, “Autoassign / Identity Strategy Caveats” \[458\]](#)
- `AutoAssignSequenceName`: The global name of the sequence that Kodo will assume to hold the value of primary key value for rows that use auto-increment. If left unset, Kodo will use a the sequence named "SEQ_<table name>".
- `MaxEmbeddedBlobSize`: Oracle is unable to persist BLOBs using the embedded update method when BLOBs get over a certain size. The size depends on database configuration, e.g. encoding. This property defines the maximum size BLOB to persist with the embedded method. Defaults to 4000 bytes.
- `MaxEmbeddedClobSize`: Oracle is unable to persist CLOBs using the embedded update method when Clobs get over a certain size. The size depends on database configuration, e.g. encoding. This property defines the maximum size CLOB to persist with the embedded method. Defaults to 4000 characters.

- `TestBatchSupport`: Some versions of the Oracle JDBC driver have problems with update counts from batch updates. Setting this property to `true` will cause Kodo to automatically test the driver's batching capabilities when it initially connects to the database and set the batching parameters and enable or disable batching as a result of the test.
- `UseSetFormOfUseForUnicode`: Prior to Oracle 10i, statements executed against unicode capable columns (the `NCHAR`, `NVARCHAR`, `NCLOB` Oracle types) required special handling to be able to store unicode values. Setting this property to `true` (the default) will cause Kodo to attempt to detect when the column is of one of these types, and if so, will attempt to correctly configure the statement using the `OraclePreparedStatement.setFormOfUse`. For more details, see the Oracle [Readme For NChar](#). Note that this can only work if Kodo is able to access the underlying `OraclePreparedStatement` instance, which may not be possible when using some third-party datasources. If Kodo detects that this is the case, a warning will be logged.

4.5. Setting the Transaction Isolation

Kodo typically retains the default transaction isolation level of the JDBC driver. However, you can specify a transaction isolation level to use through the `kodo.jdbc.TransactionIsolation` configuration property. The following is a list of standard isolation levels. Note that not all databases support all isolation levels.

- `default`: Use the JDBC driver's default isolation level. Kodo uses this option if you do not explicitly specify any other.
- `none`: No transaction isolation.
- `read-committed`: Dirty reads are prevented; non-repeatable reads and phantom reads can occur.
- `read-uncommitted`: Dirty reads, non-repeatable reads and phantom reads can occur.
- `repeatable-read`: Dirty reads and non-repeatable reads are prevented; phantom reads can occur.
- `serializable`: Dirty reads, non-repeatable reads, and phantom reads are prevented.

Example 4.8. Specifying a Transaction Isolation

JPA XML format:

```
<property name="kodo.jdbc.TransactionIsolation" value="repeatable-read"/>
```

JDO properties format:

```
kodo.jdbc.TransactionIsolation: repeatable-read
```

4.6. Setting the SQL Join Syntax

Object queries often involve using SQL joins behind the scenes. You can configure Kodo to use either SQL 92-style join syntax, in which joins are placed in the SQL `FROM` clause, the traditional join syntax, in which join criteria are part of the `WHERE` clause, or a database-specific join syntax mandated by the `DBDictionary`. Kodo only supports outer joins when using SQL 92

syntax or a database-specific syntax with outer join support.

The `kodo.jdbc.DBDictionary` plugin accepts the `JoinSyntax` property to set the system's default syntax. The available values are:

- `traditional`: Traditional SQL join syntax; outer joins are not supported.
- `database`: The database's native join syntax. Databases that do not have a native syntax will default to one of the other options.
- `sql92`: ANSI SQL92 join syntax. Outer joins are supported. Not all databases support this syntax.

You can change the join syntax at runtime through the Kodo fetch configuration API, which is described in [Chapter 9, Runtime Extensions \[538\]](#)

Example 4.9. Specifying the Join Syntax Default

JPA XML format:

```
<property name="kodo.jdbc.DBDictionary" value="JoinSyntax=sql92"/>
```

JDO properties format:

```
kodo.jdbc.DBDictionary: JoinSyntax=sql92
```

Example 4.10. Specifying the Join Syntax at Runtime

JPA:

```
import kodo.persistence.jdbc.*;
...
Query q = em.createQuery ("select m from Magazine m where m.title = 'JDJ'");
KodoQuery kq = KodoPersistence.cast (q);
JDBCFetchPlan fetch = (JDBCFetchPlan) kq.getFetchPlan ();
fetch.setJoinSyntax (JDBCFetchPlan.JOIN_SYNTAX_SQL92);
List results = q.getResultList ();
```

JDO:

```
import kodo.jdo.jdbc.*;
...
Query q = pm.newQuery (Magazine.class, "title == 'JDJ'");
JDBCFetchPlan fetch = (JDBCFetchPlan) q.getFetchPlan ();
fetch.setJoinSyntax (JDBCFetchPlan.JOIN_SYNTAX_SQL92);
List results = (List) q.execute ();
```

4.7. Accessing Multiple Databases

Through the properties we've covered thus far, you can configure each `EntityManagerFactory` or `PersistenceManagerFactory` to access a different database. If your application accesses multiple databases, we recommend that you maintain a separate properties file for each one. This will allow you to easily load the appropriate resource for each database at runtime, and to give the correct configuration file to Kodo's command-line tools during development.

4.8. Configuring the Use of JDBC Connections

In its default configuration, Kodo obtains JDBC connections on an as-needed basis. Kodo `EntityManagers` and `PersistenceManagers` do not retain a connection to the database unless they are in a datastore transaction or there are open `Extent` iterators or `Query` results that are using a live JDBC result set. At all other times, including during optimistic transactions, `EntityManagers` and `PersistenceManagers` request a connection for each query, then immediately release the connection back to the pool.

In some cases, it may be more efficient to retain connections for longer periods of time. You can configure Kodo's use of JDBC connections through the `kodo.ConnectionRetainMode` configuration property. The property accepts the following values:

- `always`: Each `EntityManager` or `PersistenceManager` obtains a single connection and uses it until the `EntityManager` or `PersistenceManager` closes.
- `transaction`: A connection is obtained when each transaction begins (optimistic or datastore), and is released when the transaction completes. Non-transactional connections are obtained on-demand.
- `on-demand`: Connections are obtained only when needed. This option is equivalent to the `transaction` option when datastore transactions are used. For optimistic transactions, though, it means that a connection will be retained only for the duration of the datastore flush and commit process.

You can also specify the connection retain mode of individual `EntityManagers` and `PersistenceManagers` when you retrieve them from their respective factories. See [Section 9.2.1, “KodoEntityManagerFactory” \[540\]](#) and [Section 9.3.1, “KodoPersistenceManagerFactory” \[541\]](#) for details.

The `kodo.FlushBeforeQueries` configuration property controls another aspect of connection usage: whether to flush transactional changes before executing object queries. This setting only applies to queries that would otherwise have to be executed in-memory because the `IgnoreChanges` property is set to false and the query may involve objects that have been changed in the current transaction. Legal values are:

- `true`: Always flush rather than executing the query in-memory. If the current transaction is optimistic, Kodo will begin a non-locking datastore transaction. This is the default.
- `false`: Never flush before a query.
- `with-connection`: Flush only if the `EntityManager` or `PersistenceManager` has already established a dedicated connection to the datastore, otherwise execute the query in-memory.

This option is useful if you use long-running optimistic transactions and want to ensure that these transactions do not consume database resources until commit. Kodo's behavior with this option is dependent on the transaction status and mode, as well as the configured connection retain mode described earlier in this section.

The flush mode can also be varied at runtime using the Kodo fetch configuration API, discussed in [Chapter 9, Runtime Extensions \[538\]](#)

The table below describes the behavior of automatic flushing in various situations. In all cases, flushing will only occur if Kodo detects that you have made modifications in the current transaction that may affect the query's results.

Table 4.2. Kodo Automatic Flush Behavior

	FlushBeforeQueries = false	FlushBeforeQueries = true	FlushBeforeQueries = with-connection; ConnectionRetainMode = on-demand	FlushBeforeQueries = with-connection; ConnectionRetainMode = transaction or always
IgnoreChanges = true	no flush	no flush	no flush	no flush
IgnoreChanges = false; no tx active	no flush	no flush	no flush	no flush
IgnoreChanges = false; datastore tx active	no flush	flush	flush	flush
IgnoreChanges = false; optimistic tx active	no flush	flush	no flush unless flush has already been invoked	flush

Example 4.11. Specifying Connection Usage Defaults

JPA XML format:

```
<property name="kodo.ConnectionRetainMode" value="on-demand"/>
<property name="kodo.FlushBeforeQueries" value="true"/>
```

JDO properties format:

```
kodo.ConnectionRetainMode: on-demand
kodo.FlushBeforeQueries: true
```

Example 4.12. Specifying Connection Usage at Runtime

JPA:

```
import kodo.persistence.*;
...
// obtaining an em with a certain transaction and connection retain mode
KodoEntityManagerFactory kemf = KodoPersistence.cast (emf);
EntityManager em = kemf.createEntityManager (PersistenceContextType.EXTENDED,
false, KodoEntityManager.CONN_RETAIN_ALWAYS);
...
// changing the flush mode for an individual EntityManager
KodoEntityManager em = KodoPersistence.cast (em);
FetchPlan fetch = kem.getFetchPlan ();
fetch.setFlushBeforeQueries (FetchPlan.QUERY_FLUSH_TRUE);
```

JDO:

```
import kodo.jdo.*;
...
// obtaining a pm with a certain transaction and connection retain mode
KodoPersistenceManagerFactory kpmf = KodoJDOHelper.cast (pmf);
PersistenceManager pm = kpmf.getPersistenceManager (false,
    KodoPersistenceManager.CONN_RETAIN_ALWAYS);
...
// changing the flush mode for an individual PersistenceManager
KodoFetchPlan fetch = (KodoFetchPlan) pm.getFetchPlan ();
fetch.setFlushBeforeQueries (KodoFetchPlan.QUERY_FLUSH_TRUE);
```

4.9. Statement Batching

In addition to connection pooling and prepared statement caching, Kodo employs statement batching to speed up JDBC updates. Statement batching is enabled by default for any JDBC driver that supports it. When batching is on, Kodo automatically orders its SQL statements to maximize the size of each batch. This can result in large performance gains for transactions that modify a lot of data.

Important

This feature requires **Kodo Enterprise Edition** or **Kodo Standard Edition** with the **Performance Pack**.

You configure statement batching through the system **DBDictionary**, which is controlled by the **kodo.jdbc.DBDictionary** configuration property. The example below shows how to enable and disable statement batching via your configuration properties.

Example 4.13. Configuring SQL Batching

The batch limit is the maximum number of statements Kodo will ever batch together. A value of -1 means "no limit" - this is the default for most dictionaries. A value of 0 disables batching.

JPA XML format:

```
<property name="kodo.jdbc.DBDictionary" value="BatchLimit=25" />
```

JDO properties format:

```
kodo.jdbc.DBDictionary: BatchLimit=25
```

4.10. Large Result Sets

By default, Kodo uses standard forward-only JDBC result sets, and completely instantiates the results of database queries on execution. When using a JDBC driver that supports version 2.0 or higher of the JDBC specification, however, you can configure Kodo to use scrolling result sets that may not bring all results into memory at once. You can also configure the number of result objects Kodo keeps references to, allowing you to traverse potentially enormous amounts of data without exhausting JVM memory.

Note

You can also configure on-demand loading for individual collection and map fields via large result set proxies. See [Section 5.5.4.2, “Large Result Set Proxies” \[461\]](#)

Use the following properties to configure Kodo's handling of result sets:

- **kodo.FetchBatchSize**: The number of objects to instantiate at once when traversing a result set. This number will be set as the fetch size on JDBC `Statement` objects used to obtain result sets. It also factors in to the number of objects Kodo will maintain a hard reference to when traversing a query result.

The fetch size defaults to -1, meaning all results will be instantiated immediately on query execution. A value of 0 means to use the JDBC driver's default batch size. Thus to enable large result set handling, you must set this property to 0 or to a positive number.

- **kodo.jdbc.ResultSetType**: The type of result set to use when executing database queries. This property accepts the following values, each of which corresponds exactly to the same-named `java.sql.ResultSet` constant:
 - `forward-only`: This is the default.
 - `scroll-sensitive`
 - `scroll-insensitive`

Different JDBC drivers treat the different result set types differently. Not all drivers support all types.

- **kodo.jdbc.FetchDirection**: The expected order in which you will access the query results. This property affects the type of datastructure Kodo will use to hold the results, and is also given to the JDBC driver in case it can optimize for certain access patterns. This property accepts the following values, each of which corresponds exactly to the same-named `java.sql.ResultSet` `FETCH` constant:

- `forward`: This is the default.
- `reverse`
- `unknown`

Not all drivers support all fetch directions.

- **kodo.jdbc.LRSSize**: The strategy Kodo will use to determine the size of result sets. This property is **only** used if you change the fetch batch size from its default of -1, so that Kodo begins to use on-demand result loading. Available values are:
 - `query`: This is the default. The first time you ask for the size of a query result, Kodo will perform a `SELECT COUNT(*)` query to determine the number of expected results. Note that depending on transaction status and settings, this can mean that the reported size is slightly different than the actual number of results available.
 - `last`: If you have chosen a scrollable result set type, this setting will use the `ResultSet.last` method to move to the last element in the result set and get its index. Unfortunately, some JDBC drivers will bring all results into memory in order to access the last one. Note that if you do not choose a scrollable result set type, then this will behave exactly like `unknown`. The default result set type is `forward-only`, so you must change the result set type in order for this property to have an effect.

- unknown: Under this setting Kodo will return `Integer.MAX_VALUE` as the size for any query result that uses on-demand loading.

Example 4.14. Specifying Result Set Defaults

JPA XML format:

```
<property name="kodo.FetchBatchSize" value="20"/>
<property name="kodo.jdbc.ResultSetType" value="scroll-insensitive"/>
<property name="kodo.jdbc.FetchDirection" value="forward"/>
<property name="kodo.jdbc.LRSSize" value="last"/>
```

JDO properties format:

```
kodo.FetchBatchSize: 20
kodo.jdbc.ResultSetType: scroll-insensitive
kodo.jdbc.FetchDirection: forward
kodo.jdbc.LRSSize: last
```

Many **Kodo runtime components** also have methods to configure these properties on a case-by-case basis through their fetch configuration. See [Chapter 9, Runtime Extensions \[538\]](#)

Example 4.15. Specifying Result Set Behavior at Runtime

JPA:

```
import java.sql.*;
import kodo.persistence.jdbc.*;

...

Query q = em.createQuery ("select m from Magazine m where m.title = 'JDJ'");
KodoQuery kq = KodoPersistence.cast (q);
JDBCFetchPlan fetch = (JDBCFetchPlan) kq.getFetchPlan ();
fetch.setFetchSize (20);
fetch.setResultSetType (ResultSet.TYPE_SCROLL_INSENSITIVE);
fetch.setFetchDirection (ResultSet.FETCH_FORWARD);
fetch.setLRSSize (JDBCFetchPlan.SIZE_LAST);
List results = (List) q.getResultList ();
```

JDO:

```
import java.sql.*;
import kodo.jdo.jdbc.*;

...

Query q = pm.newQuery (Magazine.class, "title == 'JDJ'");
JDBCFetchPlan fetch = (JDBCFetchPlan) q.getFetchPlan ();
fetch.setFetchSize (20);
fetch.setResultSetType (ResultSet.TYPE_SCROLL_INSENSITIVE);
fetch.setFetchDirection (ResultSet.FETCH_FORWARD);
fetch.setLRSSize (JDBCFetchPlan.SIZE_LAST);
```

```
List results = (List) q.execute ();
```

4.11. Default Schema

It is common to duplicate a database model in multiple schemas. You may have one schema for development and another for production, or different database users may access different schemas. Kodo facilitates these patterns with the `kodo.jdbc.Schema` configuration property. This property establishes a default schema for any unqualified table names, allowing you to leave schema names out of your mapping definitions.

The `Schema` property also establishes the default schema for new tables created through Kodo tools, such as the mapping tool covered in [Section 7.1, “Forward Mapping” \[489\]](#)

4.12. Schema Reflection

Kodo needs information about your database schema for two reasons. First, it can use schema information at runtime to validate that your schema is compatible with your persistent class definitions. Second, Kodo requires schema information during development so that it can manipulate the schema to match your object model. Kodo uses the `SchemaFactory` interface to provide runtime mapping information, and the `SchemaTool` for development-time data. Each is presented below.

4.12.1. Schemas List

By default, schema reflection acts on all the schemas your JDBC driver can "see". You can limit the schemas and tables Kodo acts on with the `kodo.jdbc.Schemas` configuration property. This property accepts a comma-separated list of schemas and tables. To list a schema, list its name. To list a table, list its full name in the form `<schema-name>.<table-name>`. If a table does not have a schema or you do not know its schema, list its name as `.<table-name>` (notice the preceding '.'). For example, to list the `BUSOBSJS` schema, the `ADDRESS` table in the `GENERAL` schema, and the `SYSTEM_INFO` table, regardless of what schema it is in, use the string:

```
BUSOBSJS,GENERAL.ADDRESS, .SYSTEM_INFO
```

Note

Some databases are case-sensitive with respect to schema and table names. Oracle, for example, requires names in all upper case.

4.12.2. Schema Factory

Kodo relies on the `kodo.jdbc.SchemaFactory` interface for runtime schema information. You can control the schema factory Kodo uses through the `kodo.jdbc.SchemaFactory` property. There are several built-in options to choose from:

- `dynamic`: This is the default setting. It is an alias for the `kodo.jdbc.schema.DynamicSchemaFactory`. The `DynamicSchemaFactory` is the most performant schema factory, because it does not validate mapping information against the database. Instead, it assumes all object-relational mapping information is correct, and dynamically builds an in-memory representation of the schema from your mapping metadata. When using this factory, it is important that your mapping metadata correctly represent your database's foreign key constraints so that Kodo can order its SQL statements to meet them.

- `native`: This is an alias for the `kodo.jdbc.schema.LazySchemaFactory`. As persistent classes are loaded by the application, Kodo reads their metadata and object-relational mapping information. This factory uses the `java.sql.DatabaseMetaData` interface to reflect on the schema and ensure that it is consistent with the mapping data being read. Because the factory doesn't reflect on a table definition until that table is mentioned by the mapping information, we call it "lazy". Use this factory if you want up-front validation that your mapping metadata is consistent with the database during development. This factory accepts the following important properties:
 - `ForeignKeys`: Set to `true` to automatically read foreign key information during schema validation.
- `table`: This is an alias for the `kodo.jdbc.schema.TableSchemaFactory`. This schema factory stores schema information as an XML document in a database table it creates for this purpose. If your JDBC driver doesn't support the `java.sql.DatabaseMetaData` standard interface, but you still want some schema validation to occur at runtime, you might use this factory. It is not recommended for most users, though, because it is easy for the stored XML schema definition to get out-of-synch with the actual database. This factory accepts the following properties:
 - `Table`: The name of the table to create to store schema information. Defaults to `KODO_SCHEMA`.
 - `PrimaryKeyColumn`: The name of the table's numeric primary key column. Defaults to `ID`.
 - `SchemaColumn`: The name of the table's string column for holding the schema definition as an XML string. Defaults to `SCHEMA_DEF`.
- `file`: This is an alias for the `kodo.jdbc.schema.FileSchemaFactory`. This factory is a lot like the `TableSchemaFactory`, and has the same advantages and disadvantages. Instead of storing its XML schema definition in a database table, though, it stores it in a file. This factory accepts the following properties:
 - `File`: The resource name of the XML schema file. By default, the factory looks for a resource called `package.schema`, located in any top-level directory of the `CLASSPATH` or in the top level of any jar in your `CLASSPATH`.

You can switch freely between schema factories at any time. The XML file format used by some factories is detailed in [Section 4.14, "XML Schema Format" \[447\]](#) As with any Kodo plugin, you can also implement your own schema factory if you have needs not met by the existing options.

4.13. Schema Tool

Most users will only access the schema tool indirectly, through the interfaces provided by other tools. You may find, however, that the schema tool is a powerful utility in its own right. The schema tool has two functions:

1. To reflect on the current database schema, optionally translating it to an XML representation for further manipulation.
2. To take in an XML schema definition, calculate the differences between the XML and the existing database schema, and apply the necessary changes to make the database match the XML.

The **XML format** used by the schema tool abstracts away the differences between SQL dialects used by different database vendors. The tool also automatically adapts its SQL to meet foreign key dependencies. Thus the schema tool is useful as a general way to manipulate schemas.

You can invoke the schema tool through the `schematool` shell/bat script included in the Kodo distribution, or through its Java class, `kodo.jdbc.schema.SchemaTool`. In addition to the universal flags of the **configuration framework**, the schema tool accepts the following command line arguments:

- `-ignoreErrors/-i <true/t | false/f>`: If `false`, an exception will be thrown if the tool encounters any database errors. Defaults to `false`.
- `-file/-f <stdout | output file>`: Use this option to write a SQL script for the planned schema modifications,

rather than committing them to the database. When used in conjunction with the `export` or `reflect` actions, the named file will be used to write the exported schema XML. If the file names a resource in the `CLASSPATH`, data will be written to that resource. Use `stdout` to write to standard output. Defaults to `stdout`.

- `-kodoTables/-kt <true/t | false/f>`: When reflecting on the schema, whether to reflect on tables and sequences whose names start with `KODO_`. Certain Kodo components may use such tables - for example, the `table` schema factory option covered in [Section 4.12.2, “Schema Factory” \[443\]](#). When using other actions, `kodoTables` controls whether these tables can be dropped. Defaults to `false`.
- `-dropTables/-dt <true/t | false/f>`: Set this option to `true` to drop tables that appear to be unused during `retain` and `refresh` actions. Defaults to `true`.
- `-dropSequences/-dsq <true/t | false/f>`: Set this option to `true` to drop sequences that appear to be unused during `retain` and `refresh` actions. Defaults to `true`.
- `-sequences/-sq <true/t | false/f>`: Whether to manipulate sequences. Defaults to `true`.
- `-indexes/-ix <true/t | false/f>`: Whether to manipulate indexes on existing tables. Defaults to `true`.
- `-primaryKeys/-pk <true/t | false/f>`: Whether to manipulate primary keys on existing tables. Defaults to `true`.
- `-foreignKeys/-fk <true/t | false/f>`: Whether to manipulate foreign keys on existing tables. Defaults to `true`.
- `-record/-r <true/t | false/f>`: Use `false` to prevent writing the schema changes made by the tool to the current **schema factory**. Defaults to `true`.
- `-schemas/-s <schema list>`: A list of schema and table names that Kodo should access during this run of the schema tool. This is equivalent to setting the `kodo.jdbc.Schemas` property for a single run.

The schema tool also accepts an `-action` or `-a` flag. The available actions are:

- `add`: This is the default action if you do not specify one. It brings the schema up-to-date with the given XML document by adding tables, columns, indexes, etc. This action never drops any schema components.
- `retain`: Keep all schema components in the given XML definition, but drop the rest from the database. This action never adds any schema components.
- `drop`: Drop all schema components in the schema XML. Tables will only be dropped if they would have 0 columns after dropping all columns listed in the XML.
- `refresh`: Equivalent to `retain`, then `add`.
- `build`: Generate SQL to build a schema matching the one in the given XML file. Unlike `add`, this option does not take into account the fact that part of the schema defined in the XML file might already exist in the database. Therefore, this action is typically used in conjunction with the `-file` flag to write a SQL script. This script can later be used to recreate the schema in the XML.
- `reflect`: Generate an XML representation of the current database schema.
- `createDB`: Generate SQL to re-create the current database. This action is typically used in conjunction with the `-file` flag to write a SQL script that can be used to recreate the current schema on a fresh database.
- `dropDB`: Generate SQL to drop the current database. Like `createDB`, this action can be used with the `-file` flag to script a database drop rather than perform it.
- `import`: Import the given XML schema definition into the current schema factory. Does nothing if the factory does not store

a record of the schema.

- `export`: Export the current schema factory's stored schema definition to XML. May produce an empty file if the factory does not store a record of the schema.

Note

The schema tool manipulates tables, columns, indexes, constraints, and sequences. It cannot create or drop the database schema objects in which the tables reside, however. If your XML documents refer to named database schemas, those schemas must exist.

We present some examples of schema tool usage below.

Example 4.16. Schema Creation

Add the necessary schema components to the database to match the given XML document, but don't drop any data:

```
schematool targetSchema.xml
```

Example 4.17. SQL Scripting

Repeat the same action as the first example, but this time don't change the database. Instead, write any planned changes to a SQL script:

```
schematool -f script.sql targetSchema.xml
```

Write a SQL script that will re-create the current database:

```
schematool -a createDB -f script.sql
```

Example 4.18. Schema Drop

Drop the current database:

```
schematool -a dropDB
```

Example 4.19. Schema Reflection

Write an XML representation of the current schema to file `schema.xml`.

```
schematool -a reflect -f schema.xml
```

4.14. XML Schema Format

The **schema tool** and **schema factories** all use the same XML format to represent database schema. The Document Type Definition (DTD) for schema information is presented below, followed by examples of schema definitions in XML.

```
<!ELEMENT schemas (schema)+>
<!ELEMENT schema (table|sequence)+>
<!ATTLIST schema name CDATA #IMPLIED>

<!ELEMENT sequence EMPTY>
<!ATTLIST sequence name CDATA #REQUIRED>
<!ATTLIST sequence initial-value CDATA #IMPLIED>
<!ATTLIST sequence increment CDATA #IMPLIED>
<!ATTLIST sequence allocate CDATA #IMPLIED>

<!ELEMENT table (column|index|pk|fk)+>
<!ATTLIST table name CDATA #REQUIRED>

<!ELEMENT column EMPTY>
<!ATTLIST column name CDATA #REQUIRED>
<!ATTLIST column type (array | bigint | binary | bit | blob | char | clob
    | date | decimal | distinct | double | float | integer | java_object
    | longvarbinary | longvarchar | null | numeric | other | real | ref
    | smallint | struct | time | timestamp | tinyint | varbinary | varchar)
    #REQUIRED>
<!ATTLIST column not-null (true|false) "false">
<!ATTLIST column auto-assign (true|false) "false">
<!ATTLIST column default CDATA #IMPLIED>
<!ATTLIST column size CDATA #IMPLIED>
<!ATTLIST column decimal-digits CDATA #IMPLIED>

<!-- the type-name attribute can be used when you want Kodo to -->
<!-- use a particular SQL type declaration when creating the -->
<!-- column. It is up to you to ensure that this type is -->
<!-- compatible with the JDBC type used in the type attribute. -->
<!ATTLIST column type-name CDATA #IMPLIED>

<!-- the 'column' attribute of indexes, pks, and fks can be used -->
<!-- when the element has only one column (or for foreign keys, -->
<!-- only one local column); in these cases the on/join child -->
<!-- elements can be omitted -->
<!ELEMENT index (on)*>
<!ATTLIST index name CDATA #REQUIRED>
<!ATTLIST index column CDATA #IMPLIED>
<!ATTLIST index unique (true|false) "false">

<!-- the 'logical' attribute of pks should be set to 'true' if -->
<!-- the primary key does not actually exist in the database, -->
<!-- but the given column should be used as a primary key for -->
<!-- O-R purposes -->
<!ELEMENT pk (on)*>
<!ATTLIST pk name CDATA #IMPLIED>
<!ATTLIST pk column CDATA #IMPLIED>
<!ATTLIST pk logical (true|false) "false">

<!ELEMENT on EMPTY>
<!ATTLIST on column CDATA #REQUIRED>

<!-- fks with a delete-action of 'none' are similar to logical -->
<!-- pks; they do not actually exist in the database, but -->
<!-- represent a logical relation between tables (or their -->
<!-- corresponding Java classes) -->
<!ELEMENT fk (join)*>
<!ATTLIST fk name CDATA #IMPLIED>
<!ATTLIST fk deferred (true|false) "false">
<!ATTLIST fk to-table CDATA #REQUIRED>
<!ATTLIST fk column CDATA #IMPLIED>
<!ATTLIST fk delete-action (cascade|default|exception|none|null) "none">
```

```

<!ELEMENT join EMPTY>
<!ATTLIST join column CDATA #REQUIRED>
<!ATTLIST join to-column CDATA #REQUIRED>
<!ATTLIST join value CDATA #IMPLIED>

<!-- unique constraint -->
<!ELEMENT unique (on)*>
<!ATTLIST unique name CDATA #IMPLIED>
<!ATTLIST unique column CDATA #IMPLIED>
<!ATTLIST unique deferred (true|false) "false">

```

Example 4.20. Basic Schema

A very basic schema definition.

```

<schemas>
  <schema>
    <sequence name="S_ARTS"/>
    <table name="ARTICLE">
      <column name="TITLE" type="varchar" size="255" not-null="true"/>
      <column name="AUTHOR_FNAME" type="varchar" size="28">
      <column name="AUTHOR_LNAME" type="varchar" size="28">
      <column name="CONTENT" type="clob">
    </table>
    <table name="AUTHOR">
      <column name="FIRST_NAME" type="varchar" size="28" not-null="true">
      <column name="LAST_NAME" type="varchar" size="28" not-null="true">
    </table>
  </schema>
</schemas>

```

Example 4.21. Full Schema

Expansion of the above schema with primary keys, constraints, and indexes, some of which span multiple columns.

```

<schemas>
  <schema>
    <sequence name="S_ARTS"/>
    <table name="ARTICLE">
      <column name="TITLE" type="varchar" size="255" not-null="true"/>
      <column name="AUTHOR_FNAME" type="varchar" size="28">
      <column name="AUTHOR_LNAME" type="varchar" size="28">
      <column name="CONTENT" type="clob">
      <pk column="TITLE"/>
      <fk to-table="AUTHOR" delete-action="exception">
        <join column="AUTHOR_FNAME" to-column="FIRST_NAME"/>
        <join column="AUTHOR_LNAME" to-column="LAST_NAME"/>
      </fk>
      <index name="ARTICLE_AUTHOR">
        <on column="AUTHOR_FNAME"/>
        <on column="AUTHOR_LNAME"/>
      </index>
    </table>
    <table name="AUTHOR">
      <column name="FIRST_NAME" type="varchar" size="28" not-null="true">
      <column name="LAST_NAME" type="varchar" size="28" not-null="true">
      <pk>
        <on column="FIRST_NAME"/>
        <on column="LAST_NAME"/>
      </pk>
    </table>
  </schema>
</schemas>

```

4.15. The SQLLine Utility

The Kodo distribution includes SQLLine, a console-based utility that interacts directly with a database using raw SQL. It is similar to other command-line database access utilities like `sqlplus` for Oracle, `mysql` for MySQL, and `isql` for Sybase/SQL Server. SQLLine can be useful as debugging tool by enabling low-level SQL interaction with the database.

SQLLine is open-source software developed by BEA employees. For complete documentation, see the project home page at <http://sqlline.sourceforge.net>. The remainder of this section discusses scenarios using SQLLine to assist with understanding and developing Kodo applications.

Note

As a separate utility, BEA does not provide technical support for SQLLine.

Example 4.22. Connecting to the Database

Use the `!connect` command to connect to your database.

```
prompt$ java sqlline.SqlLine
sqlline version 1.0.1
sqlline> !connect
Usage: connect <url> <username> <password> [driver]

sqlline>
```

Example 4.23. Connecting to the Database via Properties

SQLLine accepts a Kodo JDO properties file as a startup parameter, and will use it to connect to the database:

```
prompt$ java sqlline.SqlLine kodo.properties
Connecting to jdbc:hsqldb:tutorial_database
Connected to: HSQL Database Engine (version 1.8.0)
Driver: HSQL Database Engine Driver (version 1.8.0)
Autocommit status: true
sqlline version 1.0.1

0: jdbc:hsqldb:tutorial_database>
```

Example 4.24. Listing Commands

The `!help` command lists SQLLine command dictionary.

```
0: jdbc:hsqldb:tutorial_database> !help
!all           Execute the specified SQL against all current connections
!autocommit    Set autocommit mode on or off
!batch         Start or execute a batch of statements
!brief         Set verbose mode off
!call          Execute a callable statement
!close        Close the current connection to the database
```

```
!closeall      Close all current open connections
!columns      List all the columns for the specified table
!commit       Commit the current transaction (if autocommit is off)
...
```

Example 4.25. Examining the Tutorial Schema

SQLLine has various commands to analyze the schema of the database:

```
0: jdbc:hsqldb:tutorial_database> !tables
```

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	TABLE_TYPE	REMARKS	TYPE_CA
		ANIMAL	TABLE		
		KODOSEQUENCE	TABLE		

```
0: jdbc:hsqldb:tutorial_database> !columns ANIMAL
```

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	COLUMN_NAME	DATA_TYPE	TYPE_N
		ANIMAL	TYP	12	VARCHA
		ANIMAL	ID	-5	BIGINT
		ANIMAL	VERSION	4	INTEGE
		ANIMAL	NAME0	12	VARCHA
		ANIMAL	PRICE	7	REAL

```
0: jdbc:hsqldb:tutorial_database> !primarykeys ANIMAL
```

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	COLUMN_NAME	KEY_SEQ	PK_NA
		ANIMAL	ID	1	SYS_PK_A

Example 4.26. Issuing SQL Against the Database

Any SQL statement that the database understands can be executed in SQLLine, and the results (if any) will be displayed in a customizable format. The default is a table-like display:

```
0: jdbc:hsqldb:tutorial_database> SELECT * FROM ANIMAL;
```

TYP	ID	VERSION	NAME0	PRICE
tutorial.Dog	0	0	Binney	80.0
tutorial.Dog	1	0	Fido	50.0
tutorial.Dog	2	0	Odie	30.0
tutorial.Dog	3	0	Tasha	75.0
tutorial.Dog	4	0	Rusty	25.0

```
5 rows selected (0 seconds)
```

```
0: jdbc:hsqldb:tutorial_database> DELETE FROM ANIMAL WHERE ID > 2;
```

```
2 rows affected (0.002 seconds)
```

```
0: jdbc:hsqldb:tutorial_database> SELECT * FROM ANIMAL;
```

TYP	ID	VERSION	NAME0	PRICE
tutorial.Dog	0	0	Binney	80.0
tutorial.Dog	1	0	Fido	50.0

```
| tutorial.Dog | 2 | 0 | Odie | 30.0 |  
+-----+-----+-----+-----+  
3 rows selected (0 seconds)
```

Example 4.27. Disconnecting from the Database

Use the `!quit` command to disconnect cleanly.

```
0: jdbc:hsqldb:tutorial_database> !quit  
Closing: org.hsqldb.jdbcConnection
```

Chapter 5. Persistent Classes

Chapter 4, *Entity* [18] of the JPA Overview discusses persistent class basics in JPA. **Chapter 4, *PersistenceCapable* [179]** of the JDO Overview does the same for JDO. This chapter details the persistent class features Kodo offers beyond the core JPA and JDO specifications.

5.1. Persistent Class List

Unlike many ORM products, Kodo does not need to know about all of your persistent classes at startup. Kodo discovers new persistent classes automatically as they are loaded into the JVM; in fact you can introduce new persistent classes into running applications under Kodo. However, there are certain situations in which providing Kodo with a persistent class list is helpful:

- Kodo must be able to match entity names in JPQL queries to persistent classes. Kodo automatically knows the entity names of any persistent classes already loaded into the JVM. To match entity names to classes that have not been loaded, however, you must supply a persistent class list.
- Under JDO's **application identity**, Kodo must be able to match an identity object passed to the `PersistenceManager` to the corresponding persistent class. If the persistent class hasn't been loaded into the JVM yet, Kodo will not be able to find it. One workaround for this is to load your persistent class within your identity class' static initializer, as demonstrated in **Example 4.5, “Application Identity Class” [188]** of the JDO Overview.
- When Kodo manipulates classes in a persistent inheritance hierarchy, Kodo must be aware of all the classes in the hierarchy. If some of the classes have not been loaded into the JVM yet, Kodo may not know about them, and queries may return incorrect results.

If you use JDO's **class-name** discriminator strategy, Kodo will discover all persistent classes in the hierarchy on its own by issuing a `SELECT DISTINCT` against the database discriminator column; however, this can be inefficient for a large table. Thus it may still be preferable to provide a persistent types list, as described below.

- If you configure Kodo to create the needed database schema on startup (see **Section 7.1.4, “Runtime Forward Mapping” [?]**), Kodo must know all of your persistent classes up-front.

When any of these conditions are a factor in your JPA application, use the `class`, `mapping-file`, and `jar-file` elements of JPA's standard XML format to list your persistent classes. See **Section 6.1, “persistence.xml” [63]** for details.

Under JDO, Kodo allows you to explicitly specify all of your persistent classes in the various attributes of the `kodo.MetadataFactory` configuration property. See **Section 6.2, “Metadata Factory” [480]** for details.

Note

Listing persistent classes (or their metadata or jar files) is an all-or-nothing endeavor. If your persistent class list is non-empty, Kodo will assume that any unlisted class is not persistent.

5.2. Enhancement

In order to provide optimal runtime performance, flexible lazy loading, and efficient, immediate dirty tracking, Kodo uses an *enhancer*. An enhancer is a tool that automatically adds code to your persistent classes after you have written them. The enhancer post-processes the bytecode generated by your Java compiler, adding the necessary fields and methods to implement the required persistence features. This bytecode modification perfectly preserves the line numbers in stack traces and is compatible with Java debuggers. In fact, the only change to debugging is that the persistent setter and getter methods of entity classes using property access will be prefixed with `pc` in stack traces and step-throughs. For example, if your entity has a `getId` method for persistent property `id`, and that method throws an exception, the stack trace will report the exception from method `pcgetId`. The line

numbers, however, will correctly correspond to the `getId` method in your source file.

You can add the Kodo enhancer to your build process, or use Java 1.5's new instrumentation features to transparently enhance persistent classes when they are loaded into the JVM. The following sections describe each option.

5.2.1. Enhancing at Build Time

The enhancer can be invoked at build time via the included `kodoc` script or via its Java class, `kodo.enhance.PCEnhancer`.

Note

The included `jdoc` script is a shortcut for:

```
kodoc -jdo true ...
```

You can also enhance via Ant; see [Section 14.1.2, “Enhancer Ant Task” \[609\]](#)

Example 5.1. Using the Kodo Enhancer

JPA:

```
kodoc Magazine.java
```

JDO:

```
jdoc package.jdo
```

The enhancer accepts the standard set of command-line arguments defined by the configuration framework (see [Section 2.4, “Command Line Configuration” \[394\]](#)), along with the following flags:

- `-directory/-d <output directory>`: Path to the output directory. If the directory does not match the enhanced class' package, the package structure will be created beneath the directory. By default, the enhancer overwrites the original `.class` file.
- `-jdoEnhance/-jdo <true/t | false/f>`: Whether to enhance the classes for use with the `javax.jdo.JDOHelper` as well as performing Kodo enhancement. Defaults to `false`.
- `-enforcePropertyRestrictions/-epr <true/t | false/f>`: Whether to throw an exception when it appears that a property access entity is not obeying the restrictions placed on property access. Defaults to `false`.
- `-addDefaultConstructor/-adc <true/t | false/f>`: The spec requires that all persistent classes define a no-arg constructor. This flag tells the enhancer whether to add a protected no-arg constructor to any persistent classes that don't already have one. Defaults to `true`.

- `-tmpClassLoader/-tcl <true/t | false/f>`: Whether to load persistent classes with a temporary class loader. This allows other code to then load the enhanced version of the class within the same JVM. Defaults to `true`. Try setting this flag to `false` as a debugging step if you run into class loading problems when running the enhancer.

Each additional argument to the enhancer must be one of the following:

- The full name of a class.
- The `.java` file for a class.
- The `.class` file of a class.
- A `.jdo` metadata file. The enhancer will run on each class listed in the metadata.

If you do not supply any arguments to the enhancer, it will run on the classes in your persistent class list (see [Section 5.1, “Persistent Class List” \[452\]](#)).

If you have not specified a persistent class list, the enhancer will scan your classpath for directories containing `.jdo` files, and run on all classes listed in those files.

You can run the enhancer over classes that have already been enhanced, in which case it will not further modify the class. You can also run it over classes that are not persistence-capable, in which case it will treat the class as persistence-aware. Persistence-aware classes can directly manipulate the persistent fields of persistence-capable classes.

Note that the enhancement process for subclasses introduces dependencies on the persistent parent class being enhanced. This is normally not problematic; however, when running the enhancer multiple times over a subclass whose parent class is not yet enhanced, class loading errors can occur. In the event of a class load error, simply re-compile and re-enhance the offending classes.

5.2.2. Enhancing JPA Entities on Deployment

The JEE 5 specification includes hooks to automatically enhance JPA entities when they are deployed into a container. Thus, if you are using a JEE 5-compliant application server, Kodo will enhance your entities automatically at runtime. Note that if you prefer build-time enhancement, Kodo's runtime enhancer will correctly recognize and skip pre-enhanced classes.

If your application server does not support the JEE 5 enhancement hooks, consider using the build-time enhancement described above, or the more general runtime enhancement described in the next section.

5.2.3. Enhancing at Runtime

Kodo includes a *Java agent* for automatically enhancing persistent classes as they are loaded into the JVM. Java agents are classes that are invoked prior to your application's `main` method. Kodo's agent uses JVM hooks to intercept all class loading to enhance classes that have persistence metadata before the JVM loads them.

Note

Java agents are new to Java 5; if you are using a previous Java version, you must use Kodo's **build-time enhancement** option.

Searching for metadata for every class loaded by the JVM can slow application initialization. One way to speed things up is to take advantage of the optional persistent class list described in [Section 5.1, “Persistent Class List” \[452\]](#). If you declare a persistent class list, Kodo will only search for metadata for classes in that list.

To employ the Kodo agent, invoke `java` with the `-javaagent` set to the path to your `kodo.jar` or `kodo-runtime.jar` file.

Example 5.2. Using the Kodo Agent for Runtime Enhancement

```
java -javaagent:/home/dev/kodo/lib/kodo.jar com.xyz.Main
```

You can pass settings to the agent using Kodo's plugin syntax (see [Section 2.5, “Plugin Configuration” \[395\]](#)). The agent accepts the long form of any of the standard configuration options ([Section 2.4, “Command Line Configuration” \[394\]](#)). It also accepts the following options, the first three of which correspond exactly to the same-named options of the enhancer tool described in [Section 5.2.1, “Enhancing at Build Time” \[453\]](#):

- `addDefaultConstructor`
- `jdoEnhance`
- `enforcePropertyRestrictions`
- `scanDevPath`: Boolean indicating whether to scan the classpath for persistent types if none have been configured. If you do not specify a persistent types list and do not set this option to true, Kodo will check whether each class loaded into the JVM is persistent, and enhance it accordingly. This may slow down class load times significantly.

Example 5.3. Passing Options to the Kodo Agent

```
java -javaagent:/home/dev/kodo/lib/kodo.jar=jdoEnhance=true,addDefaultConstructor=false com.xyz.Main
```

5.2.4. Serializing Enhanced Types

By default, Kodo maintains serialization compatibility between the enhanced and unenhanced versions of a class. This allows you to serialize instances between a server using Kodo and a client that does not have access to enhanced classes or Kodo libraries. In some cases, however, you can make the persist and attach processes more robust and efficient by allowing breaks in serialization compatibility. See [Section 11.1.3, “Defining the Detached Object Graph” \[578\]](#) for details.

5.3. Object Identity

Kodo supports both datastore and application JDO identity types, including single field identity. See [Section 4.5, “JDO Identity” \[186\]](#) in the JDO Overview for coverage of the JDO identity types.

The JPA specification requires you to declare one or more identity fields in your persistent classes, as with JDO's application identity type. Kodo fully supports application identity for JPA entities; however, Kodo also allows you to use datastore identity in your entities. To use datastore identity, simply do not declare any primary key fields. Kodo will use a surrogate database key to track the identity of your objects.

You can control how your JPA datastore identity value is generated through Kodo's `kodo.persistence.DataStoreId` class annotation. This annotation has `strategy` and `generator` properties that mirror the same-named properties on the standard `javax.persistence.GeneratedValue` annotation described in [Section 5.2.2, “Id” \[34\]](#) of the JPA Overview.

The functionality of Kodo's `DataStoreId` annotation is also analogous to JDO's standard `datastore-identity` mapping element, described in [Section 15.5, “Datastore Identity” \[266\]](#) of the JDO Overview.

To retrieve the identity value of a datastore identity entity, use the `KodoEntityManager.getObjectId (Object entity)` method. See [Section 9.2.2, “KodoEntityManager” \[540\]](#) for more information on the `KodoEntityManager`.

Example 5.4. JPA Datastore Identity Metadata

```
import kodo.persistence.*;

@Entity
@DataStoreId
public class LineItem
{
    ... no @Id fields declared ...
}
```

5.3.1. Datastore Identity Objects

Internally, Kodo uses the public `kodo.util.Id` class for datastore identity objects. When writing Kodo plugins, you can manipulate datastore identity objects by casting them to this class. You can also create your own `Id` instances and pass them to any internal Kodo method that expects an identity object.

In JPA, you will never see `Id` instances directly. Instead, calling `KodoEntityManager.getObjectId` on a datastore identity object will return the `Long` surrogate primary key value for that object. You can then use this value in calls to `EntityManager.find` for subsequent lookups of the same record.

In JDO, calling `JDOHelper.getObjectId` or `PersistenceManager.getObjectId` on a persistent object with datastore identity will return an `Id` instance. Remember, however, that datastore identity in JDO is meant to be opaque; if you find yourself having to use the `Id` class often, it may be a sign that you should be using application identity.

You can also use any `Number` in Kodo JDO to represent a datastore identity primary key value. The `PersistenceManager.newObjectIdInstance(Class, Object)` method will return an `Id` instance when passed a datastore identity persistent class and a `Number` representing a primary key value. And the `PersistenceManager.getObjectById(Class, Object)` method allows you to look up a datastore identity instance given its class and `Number` primary key. The JDO specification intended these methods for single field identity key values, but in Kodo they work for datastore identity values as well.

5.3.2. Application Identity Tool

If you choose to use application identity, you may want to take advantage of Kodo JPA/JDO's application identity tool. The application identity tool generates Java code implementing the identity class for any persistent type using application identity. The code satisfies all the requirements the specification places on identity classes. You can use it as-is, or simply use it as a starting point, editing it to meet your needs.

Before you can run the application identity tool on a persistent class, the class must be compiled and must have complete metadata. All primary key fields must be marked as such in the metadata.

In JPA metadata, do not attempt to specify the `@IdClass` annotation unless you are using the application identity tool to overwrite an existing identity class. Attempting to set the value of the `@IdClass` to a non-existent class will prevent your persistent class from compiling. Instead, use the `-name` or `-suffix` options described below to tell Kodo what name to give your generated identity class. Once the application identity tool has generated the class code, you can set the `@IdClass` annotation.

In JDO metadata, set the class' `objectId-class` attribute to the desired name of the generated object identity class.

The application identity tool can be invoked via the included `appidtool` shell/bat script or via its Java class, `kodo.enhance.ApplicationIdTool`.

Note

[Section 14.1.3, “Application Identity Tool Ant Task” \[609\]](#) describes the application identity tool's Ant task.

Example 5.5. Using the Application Identity Tool

```
appidtool -s Id Magazine.java
```

```
appidtool package.jdo
```

The application identity tool accepts the standard set of command-line arguments defined by the configuration framework (see [Section 2.4, “Command Line Configuration” \[394\]](#)), including code formatting flags described in [Section 2.4.1, “Code Formatting” \[395\]](#). It also accepts the following arguments:

- `-directory/-d <output directory>`: Path to the output directory. If the directory does not match the generated oid class' package, the package structure will be created beneath the directory. If not specified, the tool will first try to find the directory of the `.java` file for the persistence-capable class, and failing that will use the current directory.
- `-ignoreErrors/-i <true/t | false/f>` : If `false`, an exception will be thrown if the tool is run on any class that does not use application identity, or is not the base class in the inheritance hierarchy (recall that subclasses never define the application identity class; they inherit it from their persistent superclass).
- `-token/-t <token>`: The token to use to separate stringified primary key values in the string form of the object id. This option is only used if you have multiple primary key fields. It defaults to `":"`.
- `-name/-n <id class name>`: The name of the identity class to generate. If this option is specified, you must run the tool on exactly one class. If the class metadata already names an object id class, this option is ignored. If the name is not fully qualified, the persistent class' package is prepended to form the qualified name.
- `-suffix/-s <id class suffix>`: A string to suffix each persistent class name with to form the identity class name. This option is overridden by `-name` or by any object id class specified in metadata.

Each additional argument to the tool must be one of the following:

- The full name of a persistent class.
- The `.java` file for a persistent class.
- The `.class` file of a persistent class.
- A `.jdo` metadata file. The tool will run on each class listed in the metadata.

If you do not supply any arguments to the tool, it will act on the classes in your persistent classes list (see [Section 5.1, “Persistent Class List” \[452\]](#)).

If you do not have a persistent classes list, the tool will scan your classpath for directories containing `.jdo` files, and run on all classes listed in those files.

5.3.3. Autoassign / Identity Strategy Caveats

[Section 5.2.3, “Generated Value” \[34\]](#) explains how to use JPA's `IDENTITY` generation type. [Section 15.5, “Datastore Identity” \[266\]](#) and [Section 15.11.3, “Automatic Values” \[291\]](#) of the JDO Overview explain how to use JDO's `autoassign` and `identity` strategies for datastore identity and field values. However, here are some additional caveats you should be aware of when using these strategies:

1. Your database must support auto-increment / identity columns, or some equivalent (see [Section 4.4.3, “OracleDictionary Properties” \[435\]](#) for how to configure a combination of triggers and sequences to fake auto-increment support in Oracle).
2. Auto-increment / identity columns must be an integer or long integer type.
3. Databases support auto-increment / identity columns to varying degrees. Some do not support them at all. Others only allow a single such column per table, and require that it be the primary key column. More lenient databases may allow non-primary key auto-increment columns, and may allow more than one per table. See your database documentation for details.
4. Statements inserting into tables with auto-increment / identity columns cannot be batched. After each insert, Kodo must go back to the database to retrieve the last inserted auto-increment value to set back in the persistent object. This can have a negative impact on performance.

5.4. Managed Inverses

Bidirectional relations are an essential part of data modeling. [Chapter 12, Mapping Metadata \[118\]](#) in the JPA Overview explains how to use the `mappedBy` annotation attribute to form bidirectional relations that also share datastore storage in JPA. [Chapter 15, Mapping Metadata \[260\]](#) in the JDO Overview explains how to do the same thing using the `mapped-by` mapping attribute in JDO.

Kodo also allows you to define purely logical bidirectional relations. The `kodo.persistence.InverseLogical` annotation names a logical inverse in JPA metadata. The `inverse-logical` field extension names a logical inverse in JDO metadata.

Example 5.6. Specifying Logical Inverses

`Magazine.coverPhoto` and `Photograph.mag` are each mapped to different foreign keys in their respective tables, but form a logical bidirectional relation. Only one of the fields needs to declare the other as its logical inverse, though it is not an error to set the logical inverse of both fields.

JPA:

```
import kodo.persistence.*;

@Entity
public class Magazine
{
    @OneToOne
    private Photograph coverPhoto;

    ...
}

@Entity
public class Photograph
```

```
{
  @OneToOne
  @InverseLogical("coverPhoto")
  private Magazine mag;
  ...
}
```

JDO:

```
<class name="Magazine">
  <field name="coverPhoto"/>
  ...
</class>
<class name="Photograph">
  <field name="mag">
    <extension vendor-name="kodo" key="inverse-logical" value="coverPhoto"/>
  </field>
  ...
</class>
```

Java does not provide any native facilities to ensure that both sides of a bidirectional relation remain consistent. Whenever you set one side of the relation, you must manually set the other side as well.

By default, Kodo behaves the same way. Kodo does not automatically propagate changes from one field in bidirectional relation to the other field. This is in keeping with the philosophy of transparency, and also provides higher performance, as Kodo does not need to analyze your object graph to correct inconsistent relations.

If convenience is more important to you than strict transparency, however, you can enable inverse relation management in Kodo. Set the **kodo.InverseManager** plugin property to `true` for standard management. Under this setting, Kodo detects changes to either side of a bidirectional relation (logical or physical), and automatically sets the other side appropriately on flush.

Example 5.7. Enabling Managed Inverses

JPA XML format:

```
<property name="kodo.InverseManager" value="true"/>
```

JDO properties format:

```
kodo.InverseManager: true
```

The inverse manager has options to log a warning or throw an exception when it detects an inconsistent bidirectional relation, rather than correcting it. To use these modes, set the manager's `Action` property to `warn` or `exception`, respectively.

By default, Kodo excludes **large result set fields** from management. You can force large result set fields to be included by setting the `ManageLRS` plugin property to `true`.

Example 5.8. Log Inconsistencies

JPA XML format:

```
<property name="kodo.InverseManager" value="true(Action=warn)"/>
```

JDO properties format:

```
kodo.InverseManager: true(Action=warn)
```

5.5. Persistent Fields

Kodo enhances the specification's support for persistent fields in many ways. This section documents aspects of Kodo's persistent field handling that may affect the way you design your persistent classes.

5.5.1. Restoring State

While the JPA specification says that you should not use rolled back objects, such objects are perfectly valid in Kodo. You can control whether the objects' managed state is rolled back to its pre-transaction values with the `kodo.RestoreState` configuration property. `none` does not roll back state (the object becomes hollow, and will re-load its state the next time it is accessed), `immutable` restores immutable values (primitives, primitive wrappers, strings) and clears mutable values so that they are re-loaded on next access, and `all` restores all managed values to their pre-transaction state.

JDO requires that all immutable fields be restored to their pre-transaction state when a transaction rollback occurs. Kodo also has the ability to restore the state of mutable fields including collections, maps, and arrays. To have the state of mutable fields restored on rollback, set the `kodo.RestoreState` or `javax.jdo.option.RestoreValues` configuration property to `all`.

5.5.2. Typing and Ordering

When loading data into a field, Kodo examines the value you assign the field in your declaration code or in your no-args constructor. If the field value's type is more specific than the field's declared type, Kodo uses the value type to hold the loaded data. Kodo also uses the comparator you've initialized the field with, if any. Therefore, you can use custom comparators on your persistent field simply by setting up the comparator and using it in your field's initial value.

Kodo also attempts to preserve the Java type and order of all collections declared in your JDO metadata. By default, Kodo uses an ordering column to maintain sequencing for all fields declared to be a `List` or array type. See [Section 7.1.3, “JDO Forward Mapping Hints” \[494\]](#) for how to prevent the creation of ordering columns during forward mapping.

Kodo does not automatically add an ordering column to `List` and array fields under JPA metadata. When you use JPA metadata, Kodo respects JPA's default of leaving collections and arrays without database sequencing.

Example 5.9. Using Initial Field Values

```
public class Company
```

```

{
  // Kodo will detect the custom comparator in the initial field value
  // and use it whenever loading data from the database into this field
  private Collection employeesBySal = new TreeSet (new SalaryComparator ());
  private Map departments;

  public Company
  {
    // or we can initialize fields in our no-args constructor; even though
    // this field is declared type Map, Kodo will detect that it's actually
    // a TreeMap and use natural ordering for loaded data
    departments = new TreeMap ();
  }

  // rest of class definition...
}

```

5.5.3. Calendar Fields and TimeZones

Kodo's support for the `java.util.Calendar` type will store only the `Date` part of the field, not the `TimeZone` associated with the field. When loading the date into the `Calendar` field, Kodo will use the `TimeZone` that was used to initialize the field.

Note

Kodo will automatically track changes made via modification methods in fields of type `Calendar`, with one exception: when using Java version 1.3, the `set()` method cannot be overridden, so when altering the calendar using that method, the field must be explicitly marked as dirty. This limitation does not apply when running with Java version 1.4 and higher.

5.5.4. Proxies

At runtime, the values of all mutable second class object fields in persistent and transactional objects are replaced with implementation-specific proxies. On modification, these proxies notify their owning instance that they have been changed, so that the appropriate updates can be made on the datastore.

5.5.4.1. Smart Proxies

Most proxies only track whether or not they have been modified. Smart proxies for collection and map fields, however, keep a record of which elements have been added, removed, and changed. This record enables the Kodo runtime to make more efficient database updates on these fields.

When designing your persistent classes, keep in mind that you can optimize for Kodo smart proxies by using fields of type `java.util.Set`, `java.util.TreeSet`, and `java.util.HashSet` for your collections whenever possible. Smart proxies for these types are more efficient than proxies for `Lists`. You can also design your own smart proxies to further optimize Kodo for your usage patterns. See the section on **custom proxies** for details.

5.5.4.2. Large Result Set Proxies

Under standard ORM behavior, traversing a persistent collection or map field brings the entire contents of that field into memory. Some persistent fields, however, might represent huge amounts of data, to the point that attempting to fully instantiate them can overwhelm the JVM or seriously degrade performance.

Kodo uses special proxy types to represent these "large result set" fields. Kodo's large result set proxies do not cache any data in memory. Instead, each operation on the proxy offloads the work to the database and returns the proper result. For example, the `contains` method of a large result set collection will perform a `SELECT COUNT(*)` query with the proper `WHERE` conditions to find out if the given element exists in the database's record of the collection. Similarly, each time you obtain an iterator Kodo performs the proper query using the current **large result set settings**, as discussed in the **JDBC** chapter. As you invoke

`Iterator.next`, Kodo instantiates the result objects on-demand.

You can free the resources used by a large result set iterator by passing it to the static `KodoPersistence.close` or `KodoJDOHelper.close` method.

Example 5.10. Using a Large Result Set Iterator

JPA:

```
import kodo.persistence.*;
...
Collection employees = company.getEmployees (); // employees is a lrs collection
Iterator itr = employees.iterator ();
while (itr.hasNext ())
    process ((Employee) itr.next ());
KodoPersistence.close (itr);
```

JDO:

```
import kodo.jdo.*;
...
Collection employees = company.getEmployees (); // employees is a lrs collection
Iterator itr = employees.iterator ();
while (itr.hasNext ())
    process ((Employee) itr.next ());
KodoJDOHelper.close (itr);
```

You can also add and remove from large result set proxies, just as with standard fields. Kodo keeps a record of all changes to the elements of the proxy, which it uses to make sure the proper results are always returned from collection and map methods, and to update the field's database record on commit.

In order to use large result set proxies in JPA, add the `kodo.persistence.LRS` annotation to the persistent field. In order to use large result set proxies in JDO, set the each large result set field's `lrs` metadata extension to `true`.

The following restrictions apply to large result set fields:

- The field must be declared as either a `java.util.Collection` or `java.util.Map`. It cannot be declared as any other type, including any sub-interface of collection or map, or any concrete collection or map class.
- The field cannot have an externalizer (see [Section 5.5.5, “Externalization” \[464\]](#))
- Because they rely on their owning object for context, large result set proxies cannot be transferred from one persistent field to another. The following code would result in an error on commit:

```
Collection employees = company.getEmployees () // employees is a lrs collection
company.setEmployees (null);
anotherCompany.setEmployees (employees);
```

Example 5.11. Marking a Large Result Set Field

JPA:

```
import kodo.persistence.*;

@Entity
public class Company
{
    @ManyToMany
    @LRS private Collection<Employee> employees;

    ...
}
```

JDO:

```
<class name="Company">
  <field name="employees">
    <collection element-type="Employee"/>
    <extension vendor-name="kodo" key="lrs" value="true"/>
  </field>
  ...
</class>
```

5.5.4.3. Custom Proxies

Kodo manages proxies through the `kodo.util.ProxyManager` interface. Kodo includes a default proxy manager, the `kodo.util.ProxyManagerImpl` (with a plugin alias name of `default`), that will meet the needs of most users. The default proxy manager understands the following configuration properties:

- `TrackChanges`: Whether to use **smart proxies**. Defaults to `true`.
- `AssertAllowedType`: Whether to check elements of maps and collections as they are added to make sure they are of the type defined in your metadata. Defaults to `false`.

For custom behavior, Kodo allows you to define your own proxy classes, and your own proxy manager. See the `kodo.util` package **Javadoc** for details on the interfaces involved, and the utility classes Kodo provides to assist you.

You can plug your custom proxy manager into the Kodo runtime through the `kodo.ProxyManager` configuration property. **Section 1.3.6, “Custom Proxies”** [621] describes proxy samples that ship with Kodo.

Example 5.12. Configuring the Proxy Manager

JPA XML format:

```
<property name="kodo.ProxyManager" value="TrackChanges=false" />
```

JDO properties format:

```
kodo.ProxyManager: TrackChanges=false
```

5.5.5. Externalization

Kodo offers the ability to write **custom field mappings** in order to have complete control over the mechanism with which fields are stored, queried, and loaded from the datastore. Often, however, a custom mapping is overkill. There is often a simple transformation from a Java field value to its database representation. Thus, Kodo provides the externalization service. Externalization allows you to specify methods that will externalize a field value to its database equivalent on store and then rebuild the value from its externalized form on load.

Note

Fields of embeddable classes used for `@EmbeddedId` values in JPA cannot have externalizers.

The JPA `kodo.persistence.Externalizer` annotation or JDO `externalizer` metadata extension sets the name of a method that will be invoked to convert the field into its external form for database storage. You can specify either the name of a non-static method, which will be invoked on the field value, or a static method, which will be invoked with the field value as a parameter. Each method can also take an optional `StoreContext` parameter for access to a persistence context. The return value of the method is the field's external form. By default, Kodo assumes that all named methods belong to the field value's class (or its superclasses). You can, however, specify static methods of other classes using the format `<class-name>.<method-name>`.

Given a field of type `CustomType` that externalizes to a string, the table below demonstrates several possible externalizer methods and their corresponding metadata extensions.

Table 5.1. Externalizer Options

Method	Extension
<code>public String CustomType.toString()</code>	<p>JPA: <code>@Externalizer("toString")</code></p> <p>JDO: <code><extension vendor-name="kodo" key="externalizer" value="toString"/></code></p>
<code>public String CustomType.toString(StoreContext ctx)</code>	<p>JPA: <code>@Externalizer("toString")</code></p> <p>JDO: <code><extension vendor-name="kodo" key="externalizer" value="toString"/></code></p>
<code>public static String AnyClass.toString(CustomType ct)</code>	<p>JPA: <code>@Externalizer("AnyClass.toString")</code></p> <p>JDO: <code><extension vendor-name="kodo" key="externalizer"</code></p>

Method	Extension
	value="AnyClass.toString" />
public static String AnyClass.toString(CustomType ct, StoreContext ctx)	<p>JPA: @Externalizer("AnyClass.toString")</p> <p>JDO: <extension vendor-name="kodo" key="externalizer" value="AnyClass.toString" /></p>

The JPA `kodo.persistence.Factory` annotation or JDO `factory` metadata extension contains the name of a method that will be invoked to instantiate the field from the external form stored in the database. Specify a static method name. The method will be invoked with the externalized value and must return an instance of the field type. The method can also take an optional `StoreContext` parameter for access to a persistence context. If a factory is not specified, Kodo will use the constructor of the field type that takes a single argument of the external type, or will throw an exception if no constructor with that signature exists.

Given a field of type `CustomType` that externalizes to a string, the table below demonstrates several possible factory methods and their corresponding metadata extensions.

Table 5.2. Factory Options

Method	Extension
public CustomType(String str)	none
public static CustomType CustomType.fromString(String str)	<p>JPA: @Factory("fromString")</p> <p>JDO: <extension vendor-name="kodo" key="factory" value="fromString" /></p>
public static CustomType CustomType.fromString(String str, StoreContext ctx)	<p>JPA: @Factory("fromString")</p> <p>JDO: <extension vendor-name="kodo" key="factory" value="fromString" /></p>
public static CustomType AnyClass.fromString(String str)	<p>JPA: @Factory("AnyClass.fromString")</p> <p>JDO: <extension vendor-name="kodo" key="factory" value="AnyClass.fromString" /></p>
public static CustomType AnyClass.fromString(String str, StoreContext ctx)	<p>JPA: @Factory("AnyClass.fromString")</p> <p>JDO: <extension vendor-name="kodo" key="factory" value="AnyClass.fromString" /></p>

If your externalized field is not a standard persistent type, you must explicitly mark it persistent. In JPA, you can force a persistent field by annotating it with `kodo.persistence.Persistent` annotation. In JDO, set the field's `persistence-modifier` to `true`. If you want the field to be in the default fetch group, also set its `default-fetch-group` metadata attribute to `true`. See [Chapter 5, Metadata \[193\]](#) for complete coverage of JDO metadata.

Note

If your custom field type is mutable and is not a standard collection, map, or date class, Kodo will not be able to detect changes to the field. You must mark the field dirty manually, or create a custom field proxy.

See `KodoEntityManager.dirty` for how to mark a field dirty manually in JPA. See `JDOHelper.makeDirty` for how to mark a field dirty in JDO.

See [Section 5.5.4, “Proxies” \[461\]](#) for a discussion of proxies. [Section 1.3.6, “Custom Proxies” \[62\]](#) covers Kodo's custom proxy samples.

You can externalize a field to virtually any value that is supported by Kodo's field mappings (embedded relations are the exception; you must declare your field to be a persistence-capable type in order to embed it). This means that a field can externalize to something as simple as a primitive, something as complex as a collection or map of persistence-capable objects, or anything in between. If you do choose to externalize to a collection or map, Kodo recognizes a family of metadata extensions for specifying type information for the externalized form of your fields - see [Section 6.4.2.7, “Type” \[486\]](#). If the external form of your field is a persistence-capable object, or contains persistence-capable objects, Kodo will correctly include the objects in its persistence-by-reachability algorithms and its delete-dependent algorithms.

The example below demonstrates a few forms of externalization. See [Section 1.3.4, “Using Externalization to Persist Second Class Objects” \[620\]](#) for a working example of externalizing many different field types.

Example 5.13. Using Externalization

JPA:

```
import kodo.persistence.*;

@Entity
public class Magazine
{
    // use Class.getName and Class.forName to go to/from strings
    @Persistent
    @Externalizer("getName")
    @Factory("forName")
    private Class cls;

    // use URL.getExternalForm for externalization. no factory;
    // we can rely on the URL string constructor
    @Persistent
    @Externalizer("toExternalForm")
    private URL url;

    // use our custom methods; notice how we use the KeyType and ElementType
    // annotations to specify the metadata for our externalized map
    @Persistent
    @Externalizer("Magazine.mapFromCustomType")
    @Factory("Magazine.mapToCustomType")
    @KeyType(String.class) @ElementType(String.class)
    private CustomType customType;

    public static Map mapFromCustomType (CustomType customType)
    {
        ... logic to pack custom type into a map ...
    }

    public static CustomType mapToCustomType (Map map)
    {
        ... logic to create custom type from a map ...
    }

    ...
}
```

JDO:

```
public class Magazine
{
    private Class    cls;
    private URL     url;
    private CustomType customType;

    public static Map mapFromCustomType (CustomType customType)
    {
        ... logic to pack custom type into a map ...
    }

    public static CustomType mapToCustomType (Map map)
    {
        ... logic to create custom type from a map ...
    }

    ...
}

<class name="Magazine">
  <field name="cls" persistence-modifier="persistent"
    default-fetch-group="true">
    <!-- use Class.getName and Class.forName to go to/from strings -->
    <extension vendor-name="kodo" key="externalizer" value="getName"/>
    <extension vendor-name="kodo" key="factory" value="forName"/>
  </field>
  <field name="url" persistence-modifier="persistent"
    default-fetch-group="true">
    <!-- use URL.getExternalForm for externalization. no -->
    <!-- factory; we can rely on the URL string constructor -->
    <extension vendor-name="kodo" key="externalizer"
      value="toExternalForm"/>
  </field>
  <field name="customType" persistence-modifier="persistent">
    <!-- use our custom methods; notice how we use the -->
    <!-- key-type and value-type extensions to specify -->
    <!-- the metadata for our externalized map -->
    <extension vendor-name="kodo" key="externalizer"
      value="Magazine.mapFromCustomType"/>
    <extension vendor-name="kodo" key="factory"
      value="Magazine.mapToCustomType"/>
    <extension vendor-name="kodo" key="key-type" value="String"/>
    <extension vendor-name="kodo" key="value-type" value="String"/>
  </field>
  ...
</class>
```

You can query externalized fields using parameters. Pass in a value of the field type when executing the query. Kodo will externalize the parameter using the externalizer method named in your metadata, and compare the externalized parameter with the value stored in the database. As a shortcut, Kodo also allows you to use parameters or literals of the field's externalized type in queries, as demonstrated in the example below.

Note

Currently, queries are limited to fields that externalize to a primitive, primitive wrapper, string, or date types, due to constraints on query syntax.

Example 5.14. Querying Externalization Fields

Assume the Magazine class has the same fields as in the previous example.

JPA:

```
// you can query using parameters
Query q = em.createQuery ("select m from Magazine m where m.url = :u");
q.setParameter ("u", new URL ("http://www.solarmetric.com"));
List results = q.getResultList ();
```

```
// or as a shortcut, you can use the externalized form directly
q = em.createQuery ("select m from Magazine m where m.url = 'http://www.solarmetric.com'");
results = q.getResultList ();
```

JDO:

```
// you can query using parameters
Query q = pm.newQuery (Magazine.class, "url == :u");
List results = (List) q.execute (new URL ("http://www.solarmetric.com"));

// or as a shortcut, you can use the externalized form directly
q = pm.newQuery (Magazine.class, "url == 'http://www.solarmetric.com'");
results = (List) q.execute ();
```

5.5.5.1. External Values

Externalization often takes simple constant values and transforms them to constant values of a different type. An example would be storing a boolean field as a char, where `true` and `false` would be stored in the database as 'T' and 'F' respectively.

Kodo allows you to define these simple translations in metadata, so that the field behaves as in **full-fledged externalization** without requiring externalizer and factory methods. External values supports translation of pre-defined simple types (primitives, primitive wrappers, and Strings), to other pre-defined simple values.

Use the JPA `kodo.persistence.ExternalValues` annotation or JDO `external-values` metadata extension to define external value translations. The values are defined in a format similar to that of **configuration plugins**, except that the value pairs represent Java and datastore values. To convert the Java boolean values of `true` and `false` to the character values T and F , for example, you would use the extension value: `true=T, false=F`.

If the type of the datastore value is different from the field's type, use the JPA `kodo.persistence.Type` annotation or JDO `type` metadata extension to define the datastore type.

Example 5.15. Using External Values

This example uses external value translation to transform a string field to an integer in the database.

JPA:

```
public class Magazine
{
    @ExternalValues({"SMALL=5", "MEDIUM=8", "LARGE=10"})
    @Type(int.class)
    private String sizeWidth;

    ...
}
```

JDO:

```
public class Magazine
{
    private String sizeWidth;

    ...
}
```

```

}
<class name="Magazine">
  <field name="sizeWidth">
    <extension vendor-name="kodo" key="external-values"
      value="SMALL=5,MEDIUM=8,LARGE=10"/>
    <extension vendor-name="kodo" key="type" value="int"/>
  </field>
  ...
</class>

```

5.6. Fetch Groups

Fetch groups are sets of fields that load together. They can be used to pool together associated fields in order to provide performance improvements over standard data fetching. Specifying fetch groups allows for tuning of lazy loading and eager fetching behavior.

The JPA Overview's [Section 5.2.6.1, “Fetch Type” \[36\]](#) describes how to use JPA metadata annotations to control whether a field is fetched eagerly or lazily. Fetch groups add a dynamic aspect to this standard ability. As you will see, Kodo's JPA extensions allow you can add and remove fetch groups at runtime to vary the sets of fields that are eagerly loaded.

The JDO Overview discusses standard JDO fetch group declarations in [Section 5.5, “Fetch Group Element” \[198\]](#). [Chapter 12, FetchPlan \[254\]](#) covers runtime fetch group behavior.

5.6.1. Custom Fetch Groups

Fetch groups are a standard feature of JDO, but are not part of the JPA specification. This section describes Kodo's JPA fetch group implementation.

Kodo places any field that is eagerly loaded according to the JPA metadata rules into the built-in *default* fetch group. The default fetch group is always active; you cannot remove it at runtime. Thus fields in this group are always loaded immediately when an object is fetched from the datastore.

A field can be a member of zero or one fetch groups, including the default fetch group. That is, fields in the default fetch group cannot be in an additional fetch group, and a field cannot declare itself a member of more than one fetch group. So to place a field in a custom fetch group, you must first exclude it from eager fetching in your JPA metadata, if it does not already default to lazy loading.

When lazy-loading a field, Kodo checks to see if that field declares itself to be a member of a fetch group. If so, Kodo will load all fields in that fetch group.

Additionally, it is possible to configure a Kodo `EntityManager` or `Query` to use a particular fetch group or set of fetch groups when loading new objects, as described later in this chapter. When this is the case, Kodo loads the default fetch group plus any fields in the specified set of additional fetch groups.

You create fetch groups with the `kodo.persistence.FetchGroup` annotation. If your class only has one custom fetch group, you can place this annotation directly on the class declaration. Otherwise, use the `kodo.persistence.FetchGroups` annotation to declare an array of individual `FetchGroup` values. The `FetchGroup` annotation has the following properties:

- `String name`: The name of the fetch group. Fetch group names are global, and are expected to be shared among classes. For example, a shopping website may use a *detail* fetch group in each product class to efficiently load all the data needed to display a product's "detail" page. The website might also define a sparse *list* fetch group containing only the fields needed to display a table of products, as in a search result.

The following names are reserved for use by Kodo: `default`, `values`, `all`, `none`, and any name beginning with `jdo`,

ejb, or kodo.

- `FetchAttribute[] attributes`: The set of persistent fields or properties in the fetch group.

As you might expect, listing a `kodo.persistence.FetchAttribute` within a `FetchGroup` includes the corresponding persistent field or property in the fetch group. Each `FetchAttribute` has the following properties:

- `String name`: The name of the persistent field or property to include in the fetch group.
- `depth`: If the attribute represents a relation, the depth to which to recurse. The current fetch group will be applied to the related object when fetching it, and so on until the depth is exhausted or the related object has no relations in the current fetch group. Under the default depth of 1, the related object will be fetched, but none of its relations will be traversed, even if they are in the current fetch group. With a depth of 2, the related object will be fetched, and if it has any relations in the current fetch group, those will be fetched with a depth of 1. A depth of 0 indicates that the recursion continues until the graph is exhausted or a related object has no relations in the current fetch group.

Thus, to create a *detail* fetch group consisting of the `publisher` and `articles` relations, with the fetch group applied recursively to the related objects, use:

Example 5.16. Custom Fetch Group Metadata

```
import kodo.persistence.*;

@Entity
@FetchGroups({
    @FetchGroup(name="detail", attributes={
        @FetchAttribute(name="publisher" depth=0),
        @FetchAttribute(name="articles" depth=0)
    }),
    ...
})
public class Magazine
{
    ...
}
```

Note

Kodo currently only supports a depth of 0 for fetch attributes. This restriction will be lifted in a future release, along with the restriction limiting each attribute to a single fetch group.

5.6.2. Custom Fetch Group Configuration

You can control the default set of fetch groups with the `kodo.FetchGroups` configuration property. Set this property to a comma-separated list of fetch group names.

In JPA, Kodo's `KodoEntityManager` and `KodoQuery` extensions to the standard `EntityManager` and `Query` interfaces provide access to a `kodo.persistence.FetchPlan` object. The `FetchPlan` maintains the set of active fetch groups. It begins with the groups defined in the `kodo.FetchGroups` property, but allows you to add and remove groups for an individual `EntityManager` or `Query` through the methods below.

```
public FetchPlan addFetchGroup (String group);
public FetchPlan addFetchGroups (String... groups);
public FetchPlan addFetchGroups (Collection groups);
```

```
public FetchPlan removeFetchGroup (String group);
public FetchPlan removeFetchGroups (String... groups);
public FetchPlan removeFetchGroups (Collection groups);
public FetchPlan resetFetchGroups ();
public Collection<String> getFetchGroups ();
public void clearFetchGroups ();
```

Chapter 9, Runtime Extensions [538] details the `KodoEntityManager`, `KodoQuery`, and `FetchPlan` interfaces.

Example 5.17. Using the FetchPlan

```
import kodo.persistence.*;
...
KodoQuery kq = KodoPersistence.cast (em.createQuery (...));
kq.getFetchPlan ().addFetchGroup ("detail");
List results = kq.getResultList ();
```

In JDO, you can override the global `FetchGroups` property at runtime using the `FetchPlan` of individual `PersistenceManagers`, `Queries`, and `Extents`. See **Chapter 12, FetchPlan [254]** for details. The extended `KodoFetchPlan` interface adds an additional useful methods for manipulating fetch groups:

```
public void resetGroups ();
```

`resetGroups` resets the active fetch group set to the groups defined in the `kodo.FetchGroups` configuration property. **Chapter 9, Runtime Extensions [538]** provides more information on Kodo extensions to standard JDO interfaces.

5.6.3. Per-field Fetch Configuration

In addition to controlling fetch configuration on a per-fetch-group basis, you can configure Kodo to include particular fields in the current fetch configuration. This allows you to add individual fields that are not in the default fetch group or in any other currently-active fetch groups to the set of fields that will be eagerly loaded from the database.

JPA `FetchPlan` methods:

```
public FetchPlan addField (String field);
public FetchPlan addFields (String... fields);
public FetchPlan addFields (Class cls, String... fields);
public FetchPlan addFields (Collection fields);
public FetchPlan addFields (Class cls, Collection fields);
public FetchPlan removeField (String field);
public FetchPlan removeFields (String... fields);
public FetchPlan removeFields (Class cls, String... fields);
public FetchPlan removeFields (Collection fields);
public FetchPlan removeFields (Class cls, Collection fields);
public Collection<String> getFields ();
public void clearFields ();
```

JDO `KodoFetchPlan` methods:

```

public KodoFetchPlan setFields (String[] fields);
public KodoFetchPlan setFields (Class cls, String[] fields);
public KodoFetchPlan setFields (Collection fields);
public KodoFetchPlan setFields (Class cls, Collection fields);
public KodoFetchPlan addField (String field);
public KodoFetchPlan addField (Class cls, String field);
public KodoFetchPlan removeField (String field);
public KodoFetchPlan removeField (Class cls, String field);
public boolean hasField (Class cls, String field);
public Collection getFields ();
public KodoFetchPlan clearFields ();

```

The methods that take only string arguments use the fully-qualified field name, such as `org.mag.Magazine.publisher`. Similarly, `getFields` returns the set of fully-qualified field names. In all methods, the named field must be defined in the class specified in the invocation, not a superclass. So, if the field `publisher` is defined in base class `Publication` rather than subclass `Magazine`, you must invoke `addField (Publication.class, "publisher")` and not `addField (Magazine.class, "publisher")`. This is stricter than Java's default field-masking algorithms, which would allow the latter method behavior if `Magazine` did not also define a field called `publisher`.

In order to avoid the cost of reflection, Kodo does not perform any validation of the field name / class name pairs that you put into the fetch configuration. If you specify non-existent class / field pairs, nothing adverse will happen, but you will receive no notification of the fact that the specified configuration is not being used.

Example 5.18. Adding an Eager Field

JPA:

```

import kodo.persistence.*;
...
KodoEntityManager kem = KodoPersistence.cast (em);
kem.getFetchPlan ().addField (Magazine.class, "publisher");
Magazine mag = em.find (Magazine.class, magId);

```

JDO:

```

import kodo.jdo.*;
...
PersistenceManager pm = ...;
KodoFetchPlan fetch = (KodoFetchPlan) pm.getFetchPlan ();
fetch.addField (Magazine.class, "publisher");
Magazine mag = (Magazine) pm.getObjectById (magId);

```

5.6.4. Implementation Notes

- JPA defaults to eagerly loading direct relations to other persistence-capable objects. JDO defaults to lazily loading direct these relations. Kodo respects the defaults of the metadata format you choose to use. If you use JPA annotations for metadata, Kodo will include `@OneToOne` and `@ManyToOne` relations in the default fetch group unless you explicitly set `fetch=FetchType.LAZY`. If you use JDO metadata, Kodo will exclude direct relations from the default fetch group unless you explicitly set `default-fetch-group="true"`.

- Even when a direct relation is not eagerly fetched, Kodo selects the foreign key columns and caches the values. This way when you do traverse the relation, Kodo can often find the related object in its cache, or at least avoid joins when loading the related object from the database.

In JDO, you can force Kodo not to select the foreign key values by explicitly setting `default-fetch-group="false"` for the field in your JDO metadata.

- The above implicit foreign key-selecting behavior does not always apply when the relation is in a subclass table. If the subclass table would not otherwise be joined into the select, Kodo avoids the extra join just to select the foreign key values.

5.7. Eager Fetching

Eager fetching is the ability to efficiently load subclass data and related objects along with the base instances being queried. Typically, Kodo has to make a trip to the database whenever a relation is loaded, or when you first access data that is mapped to a table other than the least-derived superclass table. If you perform a query that returns 100 `Person` objects, and then you have to retrieve the `Address` for each person, Kodo may make as many as 101 queries (the initial query, plus one for the address of each person returned). Or if some of the `Person` instances turn out to be `Employees`, where `Employee` has additional data in its own joined table, Kodo once again might need to make extra database trips to access the additional employee data. With eager fetching, Kodo can reduce these cases to a single query.

Important

This feature requires **Kodo Enterprise Edition** or **Kodo Standard Edition** with the **Performance Pack**.

Eager fetching only affects relations in the fetch groups being loaded (see [Section 5.6, “Fetch Groups” \[469\]](#)). In other words, relations that would not normally be loaded immediately when retrieving an object or accessing a field are not affected by eager fetching. In our example above, the address of each person would only be eagerly fetched if the query were configured to include the address field or its fetch group, or if the address were in the default fetch group. This allows you to control exactly which fields are eagerly fetched in different situations. Similarly, queries that exclude subclasses aren't affected by eager subclass fetching, described below.

Eager fetching has three modes:

- `none`: No eager fetching is performed. Related objects are always loaded in an independent select statement. No joined subclass data is loaded unless it is in the table(s) for the base type being queried. Unjoined subclass data is loaded using separate select statements rather than a SQL UNION operation.
- `join`: In this mode, Kodo joins to to-one relations in the configured fetch groups. If Kodo is loading data for a single instance, then Kodo will also join to any collection field in the configured fetch groups. When loading data for multiple instances, though, (such as when executing a `Query`) Kodo will not join to collections by default. Instead, Kodo defaults to `parallel` mode for collections, as described below. You can force Kodo use a join rather than parallel mode for a collection field using the metadata extension described in [Section 7.9.2.1, “Eager Fetch Mode” \[520\]](#)

Under `join` mode, Kodo uses a left outer join (or inner join, if the relations' field metadata declares the relation non-nullable) to select the related data along with the data for the target objects. This process works recursively for to-one joins, so that if `Person` has an `Address`, and `Address` has a `TelephoneNumber`, and the fetch groups are configured correctly, Kodo might issue a single select that joins across the tables for all three classes. To-many joins can not recursively spawn other to-many joins, but they can spawn recursive to-one joins.

Under the `join` subclass fetch mode, subclass data in joined tables is selected by outer joining to all possible subclass tables of the type being queried. Unjoined subclass data is selected with a SQL UNION where possible. As you'll see below, subclass data fetching is configured separately from relation fetching, and can be disabled for specific classes.

Note

Some databases may not support UNIONS or outer joins. Also, Kodo can not use outer joins if you have set the `DBDictionary`'s `JoinSyntax` to `traditional`. See [Section 4.6, “Setting the SQL Join Syntax” \[436\]](#).

- `parallel`: Under this mode, Kodo selects to-one relations and joined collections as outlined in the `join` mode description above. Unjoined collection fields, however, are eagerly fetched using a separate select statement for each collection, executed in parallel with the select statement for the target objects. The parallel selects use the `WHERE` conditions from the primary select, but add their own joins to reach the related data. Thus, if you perform a query that returns 100 `Company` objects, where each company has a list of `Employee` objects and `Department` objects, Kodo will make 3 queries. The first will select the company objects, the second will select the employees for those companies, and the third will select the departments for the same companies. Just as for joins, this process can be recursively applied to the objects in the relations being eagerly fetched. Continuing our example, if the `Employee` class had a list of `Projects` in one of the fetch groups being loaded, Kodo would execute a single additional select in parallel to load the projects of all employees of the matching companies.

Using an additional select to load each collection avoids transferring more data than necessary from the database to the application. If eager joins were used instead of parallel select statements, each collection added to the configured fetch groups would cause the amount of data being transferred to rise dangerously, to the point that you could easily overwhelm the network.

Polymorphic to-one relations to table-per-class mappings use parallel eager fetching because proper joins are impossible. You can force other to-one relations to use parallel rather than join mode eager fetching using the metadata extension described in [Section 7.9.2.1, “Eager Fetch Mode” \[520\]](#)

Setting your subclass fetch mode to `parallel` affects table-per-class and vertical inheritance hierarchies. Under parallel mode, Kodo issues separate selects for each subclass in a table-per-class inheritance hierarchy, rather than UNIONing all subclass tables together as in join mode. This applies to any operation on a table-per-class base class: query, by-id lookup, or relation traversal.

When dealing with a vertically-mapped hierarchy, on the other hand, parallel subclass fetch mode only applies to queries. Rather than outer-joining to subclass tables, Kodo will issue the query separately for each subclass. In all other situations, parallel subclass fetch mode acts just like join mode in regards to vertically-mapped subclasses.

When Kodo knows that it is selecting for a single object only, it never uses `parallel` mode, because the additional selects can be made lazily just as efficiently. This mode only increases efficiency over `join` mode when multiple objects with eager relations are being loaded, or when multiple selects might be faster than joining to all possible subclasses.

5.7.1. Configuring Eager Fetching

You can control Kodo's default eager fetch mode through the `kodo.jdbc.EagerFetchMode` and `kodo.jdbc.SubclassFetchMode` configuration properties. Set each of these properties to one of the mode names described in the previous section: `none`, `join`, `parallel`. If left unset, the eager fetch mode defaults to `parallel` and the subclass fetch mode defaults to `join` (assuming your license supports eager fetching at all). These are generally the most robust and performant strategies.

You can easily override the default fetch modes at runtime for any lookup or query through Kodo's fetch configuration APIs. See [Chapter 9, Runtime Extensions \[538\]](#) for details.

Example 5.19. Setting the Default Eager Fetch Mode

JPA XML format:

```
<property name="kodo.jdbc.EagerFetchMode" value="parallel"/>
<property name="kodo.jdbc.SubclassFetchMode" value="join"/>
```

JDO properties format:

```
kodo.jdbc.EagerFetchMode: parallel
kodo.jdbc.SubclassFetchMode: join
```

Example 5.20. Setting the Eager Fetch Mode at Runtime

JPA:

```
import kodo.persistence.*;
import kodo.persistence.jdbc.*;

...

Query q = em.createQuery ("select p from Person p where p.address.state = 'TX'");
KodoQuery kq = KodoPersistence.cast (q);
JDBCFetchPlan fetch = (JDBCFetchPlan) kq.getFetchPlan ();
fetch.setEagerFetchMode (JDBCFetchPlan.EAGER_PARALLEL);
fetch.setSubclassFetchMode (JDBCFetchPlan.EAGER_JOIN);
List results = q.getResultList ();
```

JDO:

```
import kodo.jdo.jdbc.*;

...

Query q = pm.newQuery (Person.class, "address.state == 'TX'");
JDBCFetchPlan fetch = (JDBCFetchPlan) q.getFetchPlan ();
fetch.setEagerFetchMode (JDBCFetchPlan.EAGER_PARALLEL);
fetch.setSubclassFetchMode (JDBCFetchPlan.EAGER_JOIN);
List results = (List) q.execute ();
```

You can specify a default subclass fetch mode for an individual class with the metadata extension described in [Section 7.9.1.1, “Subclass Fetch Mode” \[520\]](#) Note, however, that you cannot "upgrade" the runtime fetch mode with your class setting. If the runtime fetch mode is none, no eager subclass data fetching will take place, regardless of your metadata setting.

This applies to the eager fetch mode metadata extension as well (see [Section 7.9.2.1, “Eager Fetch Mode” \[520\]](#)) You can use this extension to disable eager fetching on a field or to declare that a collection would rather use joins than parallel selects or vice versa. But an extension value of `join` won't cause any eager joining if the fetch configuration's setting is none.

5.7.2. Eager Fetching Considerations and Limitations

There are several important points that you should consider when using eager fetching:

- When you are using `parallel` eager fetch mode and you have large result sets enabled (see [Section 4.10, “Large Result Sets” \[440\]](#)) or you place a range on a query, Kodo performs the needed parallel selects on one page of results at a time. For example, suppose your `FetchBatchSize` is set to 20, and you perform a large result set query on a class that has collection fields in the configured fetch groups. Kodo will immediately cache the first 20 results of the query using `join` mode eager fetching only. Then, it will issue the extra selects needed to eager fetch your collection fields according to `parallel` mode. Each select will use a SQL `IN` clause (or multiple `OR` clauses if your class has a compound primary key) to limit the selected

collection elements to those owned by the 20 cached results.

Once you iterate past the first 20 results, Kodo will cache the next 20 and again issue any needed extra selects for collection fields, and so on. This pattern ensures that you get the benefits of eager fetching without bringing more data into memory than anticipated.

- Eager fetching can sometimes be *less* efficient than standard fetching when circular relations are included in the configured fetch groups.
- Once Kodo eager-joins into a class, it cannot issue any further eager to-many joins or parallel selects from that class in the same query. To-one joins, however, can recurse to any level.
- Using a to-many join makes it impossible to determine the number of instances the result set contains without traversing the entire set. This is because each result object might be represented by multiple rows. Thus, queries with a range specification or queries configured for lazy result set traversal automatically turn off eager to-many joining.
- Kodo cannot eagerly join to polymorphic relations to non-leaf classes in a table-per-class inheritance hierarchy. You can work around this restriction using the mapping extensions described in [Section 7.9.2.2, “Nonpolymorphic” \[520\]](#)

5.8. Lock Groups

Kodo supports both optimistic and datastore locking strategies, but optimistic locking is the preferred approach in most applications. Typically, optimistic locking is performed at the object level of granularity. That is, changes to any part of the same object in concurrent transactions will result in an optimistic locking exception being thrown by the transaction that commits last. In many applications, this is acceptable. However, if your application has a high likelihood of concurrent writes to different parts of the same object, then it may be advantageous to use a finer-grained optimistic lock. Additionally, certain parts of an object model may be best modeled without any locking at all, or with a last-commit-wins strategy. It is for these types of situations that Kodo offers customizable optimistic lock groups, which allow you to achieve sub-object-level locking granularity.

Important

This feature requires **Kodo Enterprise Edition** or **Kodo Standard Edition** with the **Performance Pack**.

For example, an `Employee` class may have some fields configurable by the employee the object represents (`firstName`, `lastName`, `phoneNumber`), some that are only modifiable by that employee's manager (`salary`, `title`), and some in which concurrent updates are acceptable (a list of `projects`). In such a model, you can greatly improve the success of concurrent updates in optimistic transactions by putting `firstName`, `lastName`, and `phoneNumber` into one lock group, `salary` and `title` into another, and excluding the `projects` field from optimistic lock checks altogether.

You specify a field's lock group in JPA metadata with the `kodo.persistence.LockGroup` annotation. You specify a field's lock group in JDO metadata with the `lock-group` metadata extension. See [Section 6.4.2.5, “Lock Group” \[486\]](#) for details on lock group metadata.

Example 5.21. Lock Group Metadata

JPA:

```
import kodo.persistence.*;

@Entity
public class Employee
{
    // default lock group
    private String firstName;
    private String lastName;
    private String phoneNumber;
}
```

```

// named lock group
@LockGroup("corporate") private float salary;
@LockGroup("corporate") private String title;

// no lock group; allow concurrent modifications
@LockGroup(LockGroup.NONE) private Set<Project> projects;

...
}

```

JDO:

```

public class Employee
{
    private String firstName;
    private String lastName;
    private String phoneNumber;
    private float salary;
    private String title;
    private Set projects;

    ...
}

<?xml version="1.0"?>
<jdo>
  <package name="">
    <class name="Employee">
      <!-- named lock group -->
      <field name="salary">
        <extension vendor-name="kodo" key="lock-group" value="corporate"/>
      </field>
      <field name="title">
        <extension vendor-name="kodo" key="lock-group" value="corporate"/>
      </field>
      <!-- no lock group; allow concurrent modifications -->
      <field name="projects">
        <collection element-type="Project"/>
        <extension vendor-name="kodo" key="lock-group" value="none"/>
      </field>
    </class>
  </package>
</jdo>

```

Currently, lock groups are only supported when using number and timestamp version strategies. They are not supported in the state-comparison strategy, though you can still exclude fields from participating in optimistic versioning under this strategy by setting the their lock group to none.

5.8.1. Lock Groups and Subclasses

Due to mapping restrictions, subclasses cannot simply declare additional lock groups implicitly, as is done in the example shown above. Instead, the least-derived mapped type in the persistent hierarchy must list all lock groups that its children can use via the **kodo.persistence.LockGroups** annotation in JPA metadata, or with JDO's class-level `lock-groups` metadata extension. For example, if the `Employee` class in the last example extended `Person`, the metadata would have looked like so:

Example 5.22. Lock Group Metadata

JPA:

```

import kodo.persistence.*;

@Entity
@LockGroups({"corporate"})
public class Person

```

```

{
    // default lock group
    private String firstName;
    private String lastName;
    private String phoneNumber;
}
...
@Entity
public class Employee
    extends Person
{
    // named lock group
    @LockGroup("corporate") private float salary;
    @LockGroup("corporate") private String title;

    // no lock group; allow concurrent modifications
    @LockGroup(LockGroup.NONE) private Set<Project> projects;
}
...
}

```

JDO:

```

public class Person
{
    private String firstName;
    private String lastName;
    private String phoneNumber;
}
...

public class Employee
    extends Person
{
    // these fields can only be set by the employee's manager
    private float salary;
    private String title;

    // this field might be updated concurrently by the employee,
    // other team members, or the employee's manager
    private Set projects;
}
...

<?xml version="1.0"?>
<jdo>
  <package name="">
    <class name="Person">
      <!-- here we list the lock groups that will be used by Employee -->
      <extension vendor-name="kodo" key="lock-groups" value="corporate"/>
    </class>
    <class name="Employee">
      <!-- named lock group -->
      <field name="salary">
        <extension vendor-name="kodo" key="lock-group" value="corporate"/>
      </field>
      <field name="title">
        <extension vendor-name="kodo" key="lock-group" value="corporate"/>
      </field>
      <!-- no lock group; allow concurrent modifications -->
      <field name="projects">
        <collection element-type="Project"/>
        <extension vendor-name="kodo" key="lock-group" value="none"/>
      </field>
    </class>
  </package>
</jdo>

```

The exceptions to this rule are the none and default built-in lock groups. They can be used at any point in the inheritance hierarchy without pre-declaration. Additionally, the lock groups listing can contain lock groups that would otherwise be implicitly defined in the least-derived type metadata.

5.8.2. Lock Group Mapping

When using custom lock groups with a relational database, Kodo will need a version column for each of the groups, instead of just one version column. This means that you must use surrogate versioning; you cannot use a version field. Kodo also currently requires that all the version columns for a given object be in the same table. Finally, it is only possible to use a single version strategy for a given object. That is, you cannot have one version number column and another timestamp version column.

Use the `kodo.persistence.jdbc.VersionColumn(s)` annotation to specify the version column for each lock group in JPA mapping. In your JDO mappings, use the `lock-group` column extension to specify the version column for each lock group.

Example 5.23. Mapping Lock Groups

JPA:

```
import kodo.persistence.jdbc.*;

@Entity
@Table(name="EMP")
@VersionColumns({
    @VersionColumn(name="VERS_CORP" lockGroup="corporate"),
    @VersionColumn(name="VERS")
})
public class Employee
{
    ...
}
```

JDO:

```
<class name="Employee" table="EMP">
  <version strategy="version-number">
    <column name="VERS_CORP">
      <extension vendor-name="kodo" key="lock-group" value="corporate"/>
    </column>
    <!-- column for default group doesn't need an extension -->
    <column name="VERS"/>
  </version>
  ...
</class>
```

Chapter 6. Metadata

The JPA and JDO Overview documents cover standard metadata in [Chapter 5, Metadata \[28\]](#) and [Chapter 5, Metadata \[193\]](#), respectively. This chapter discusses the tools Kodo provides to aid in metadata creation, and metadata extensions that Kodo recognizes.

6.1. Generating Default JDO Metadata

Kodo's mapping tool can generate default JDO metadata for your persistent classes. The tool can only rely on reflection, so it cannot fill in information that is not available from the class definition itself, such as the element type of collections or the primary key fields of a class using application identity. It does, however, provide a good starting point from which to build up your metadata. See [Section 7.1, “Forward Mapping” \[489\]](#) for details on the mapping tool.

Example 6.1. Generating Metadata with the Mapping Tool

This command adds metadata for all classes in the `mypackage` directory to the `mypackage/package.jdo` file. If the file does not exist, the mapping tool will create it.

```
mappingtool -a add -m true -f mypackage/package.jdo mypackage/*.java
```

6.2. Metadata Factory

The `kodo.MetaDataFactory` configuration property controls metadata loading and storing. This property takes a plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing a concrete `kodo.meta.MetaDataFactory` implementation. A metadata factory can load mapping information as well as persistence metadata, or it can leave mapping information to a separate *mapping factory* (see [Section 7.5, “Mapping Factory” \[503\]](#)). Kodo recognizes the following built-in metadata factories:

- `jpa`: Standard JPA metadata. This is an alias for the `kodo.persistence.PersistenceMetaDataFactory`.
- `jdo`: Standard JDO metadata. This is an alias for the `kodo.jdo.JDOMetaDataFactory`, which has the following configurable property:
 - `ScanTopDown`: This boolean property controls whether Kodo looks for metadata files top-down in the package tree. Kodo defaults to bottom-up scanning, meaning that when scanning for metadata for class `C`, Kodo looks first for `C.jdo`, then `package.jdo` in `C`'s package, then `package.jdo` in the parent package, and so forth to the package root.
- `kodo3`: Kodo 3.x metadata compatibility. This compatibility factory recognizes Kodo 3.x metadata extensions. It is an alias for the `kodo.jdo.DeprecatedJDOMetaDataFactory`.

The standard metadata factories all accept the following properties for locating persistent classes. Each property represents a different mechanism for locating persistent types; you can choose the mechanism or combination of mechanisms that are most convenient. See [Section 5.1, “Persistent Class List” \[452\]](#) for a discussion of when it is necessary to list your persistent classes.

- `Types`: A semicolon-separated list of fully-qualified persistent class names.
- `Resources`: A semicolon-separated list of resource paths to metadata files or jar archives. Each jar archive will be scanned for annotated JPA entities or JDO metadata files based on your metadata factory.

- **URLs:** A semicolon-separated list of URLs of metadata files or jar archives. Each jar archive will be scanned for annotated JPA entities or JDO metadata files based on your metadata factory.
- **ClasspathScan:** A semicolon-separated list of directories or jar archives listed in your classpath. Each directory and jar archive will be scanned for annotated JPA entities or JDO metadata files based on your metadata factory.

Example 6.2. Setting a Standard Metadata Factory

JPA:

```
<property name="kodo.MetadataFactory" value="jpa"/>
```

JDO:

```
kodo.MetadataFactory: jdo(Resources=com/aaa/package.jdo;com/bbb/package.jdo,\n  ClasspathScan=build)
```

Example 6.3. Setting a Custom Metadata Factory

JPA XML format:

```
<property name="kodo.MetadataFactory" value="com.xyz.CustomMetadataFactory"/>
```

JDO properties format:

```
kodo.MetadataFactory: com.xyz.CustomMetadataFactory
```

6.3. Additional JPA Metadata

This section describes Kodo's core additions to standard entity metadata. We present the object-relational mapping syntax to support these additions in [Section 7.7, “Additional JPA Mappings” \[508\]](#). Finally, [Section 6.4, “Metadata Extensions” \[483\]](#) covers additional extensions to both JDO and JPA metadata that allow you to access auxiliary Kodo features.

6.3.1. Datastore Identity

JPA typically requires you to declare one or more `Id` fields to act as primary keys. Kodo, however, can create and maintain a surrogate primary key value when you do not declare any `Id` fields. This form of persistent identity is called *datastore identity*. [Section 5.3, “Object Identity” \[455\]](#) discusses Kodo's support for datastore identity in JPA. We cover how to map your datastore

identity primary key column in [Section 7.7.1, “Datastore Identity Mapping” \[508\]](#)

6.3.2. Surrogate Version

Just as Kodo can maintain your entity's identity without any `Id` fields, Kodo can maintain your entity's optimistic version without any `Version` fields. [Section 7.7.2, “Surrogate Version Mapping” \[509\]](#) shows you how to map surrogate version columns.

6.3.3. Persistent Field Values

JPA defines `Basic`, `Lob`, `Embedded`, `ManyToOne`, and `OneToOne` persistence strategies for direct field values. Kodo supports all of these standard strategies, but adds one of its own: `Persistent`. The `kodo.persistence.Persistent` metadata annotation can represent any direct field value, including custom types. It has the following properties:

- `FetchType fetch`: Whether to load the field eagerly or lazily. Corresponds exactly to the same-named property of standard JPA annotations such as `Basic`. Defaults to `FetchType.EAGER`.
- `CascadeType[] cascade`: Array of enum values defining cascade behavior for this field. Corresponds exactly to the same-named property of standard JPA annotations such as `ManyToOne`. Defaults to empty array.
- `String mappedBy`: Names the field in the related entity that maps this bidirectional relation. Corresponds to the same-named property of standard JPA annotations such as `OneToOne`.
- `boolean optional`: Whether the value can be null. Corresponds to the same-named property of standard JPA annotations such as `ManyToOne`, but can apply to non-entity object values as well. Defaults to `true`.
- `boolean embedded`: Set this property to `true` if the field value is stored as an embedded object.

Though you can use the `Persistent` annotation in place of most of the standard direct field annotations mentioned above, we recommend primarily using it for non-standard and custom types for which no standard JPA annotation exists. For example, [Section 7.7.3, “Multi-Column Mappings” \[510\]](#) demonstrates the use of the `Persistent` annotation to denote a persistent `java.awt.Point` field.

6.3.4. Persistent Collection Fields

JPA standardizes support for collections of entities with the `OneToMany` and `ManyToMany` persistence strategies. Kodo expands collection support to handle collections of simple types (primitive wrappers, `Strings`, etc), custom types, and embedded objects.

The `kodo.persistence.PersistentCollection` metadata annotation represents a persistent collection field. It has the following properties:

- `Class elementType`: The class of the collection elements. This information is usually taken from the parameterized collection element type. You must supply it explicitly, however, if your field isn't a parameterized type.
- `FetchType fetch`: Whether to load the collection eagerly or lazily. Corresponds exactly to the same-named property of standard JPA annotations such as `Basic`. Defaults to `FetchType.LAZY`.
- `String mappedBy`: Names the field in the related entity that maps this bidirectional relation. Corresponds to the same-named property of standard JPA annotations such as `ManyToMany`.
- `CascadeType[] elementCascade`: Array of enum values defining cascade behavior for the collection elements. Corresponds exactly to the `cascade` property of standard JPA annotations such as `ManyToMany`. Defaults to empty array.
- `boolean elementEmbedded`: Set this property to `true` if the elements are stored as embedded objects.

Section 7.7.6, “Collections” [512] contains several examples of using `PersistentCollection` to mark non-standard collection fields persistent.

6.3.5. Persistent Map Fields

JPA has limited support for maps. Kodo introduces the `kodo.persistence.PersistentMap` metadata annotation to represent a persistent map field. It has the following properties:

- `Class keyType`: The class of the map keys. This information is usually taken from the parameterized map key type. You must supply it explicitly, however, if your field isn't a parameterized type.
- `Class elementType`: The class of the map values. This information is usually taken from the parameterized map value type. You must supply it explicitly, however, if your field isn't a parameterized type.
- `FetchType fetch`: Whether to load the collection eagerly or lazily. Corresponds exactly to the same-named property of standard JPA annotations such as `Basic`. Defaults to `FetchType.LAZY`.
- `CascadeType[] keyCascade`: Array of enum values defining cascade behavior for the map keys. Corresponds exactly to the `cascade` property of standard JPA annotations such as `ManyToOne`. Defaults to empty array.
- `CascadeType[] elementCascade`: Array of enum values defining cascade behavior for the map values. Corresponds exactly to the `cascade` property of standard JPA annotations such as `ManyToOne`. Defaults to empty array.
- `boolean keyEmbedded`: Set this property to `true` if the map keys are stored as embedded objects.
- `boolean elementEmbedded`: Set this property to `true` if the map values are stored as embedded objects.

Map keys and values in Kodo can be entities, simple types (primitive wrappers, `Strings`, etc), custom types, or embedded objects. **Section 7.7.8, “Maps” [515]** contains several examples of using `PersistentMap` to annotate persistent map fields.

6.4. Metadata Extensions

Kodo extends standard metadata to allow you to access advanced Kodo functionality. This section covers persistence metadata extensions; we discuss mapping metadata extensions in **Section 7.9, “Mapping Extensions” [519]**. All metadata extensions are optional; Kodo will rely on its defaults when no explicit data is provided.

In JDO metadata, Kodo extensions are expressed with standard JDO `extension` elements. All Kodo extensions use a `vendor-name` of `kodo`. The next sections present a list of the available `class` and `field` element extension keys.

6.4.1. Class Extensions

Kodo recognizes the following class extensions:

6.4.1.1. Fetch Groups

JDO standardizes fetch group definition with the `fetch-group` XML mapping element. Fetch groups, however, are not part of the JPA standard. The `kodo.persistence.FetchGroups` and `kodo.persistence.FetchGroup` annotations allow you to define fetch groups in your JPA entities. **Section 5.6, “Fetch Groups” [469]** discusses Kodo's support for fetch groups in general; see **Section 5.6.1, “Custom Fetch Groups” [469]** for how to use these annotations in particular.

6.4.1.2. Data Cache

Section 10.1, “Data Cache” [560] examines caching in Kodo. Metadata extensions allow individual classes to override system caching defaults.

Kodo JPA defines the `kodo.persistence.DataCache` annotation for caching information. This annotation has the following properties:

- `boolean enabled`: Whether to cache data for instances of the class. Defaults to `true` for base classes, or the superclass value for subclasses. If you set this property to `false`, all other properties are ignored.
- `String name`: Place data for instances of the class in a named cache. By default, instance data is placed in the same cache as superclass data, or the default cache configured through the `kodo.DataCache` configuration property for base classes.
- `int timeout`: The number of milliseconds data for the class remains valid. Use `-1` for no timeout. Defaults to the `kodo.DataCacheTimeout` property value.

Kodo JDO uses the `data-cache` and `data-cache-timeout` XML extension keys to specify class caching information. The `data-cache-timeout` key sets the number of milliseconds cached data for an instance remains valid. A value of `-1` means never to expire instance data. This extension overrides the `kodo.DataCacheTimeout` configuration property.

The `data-cache` key accepts the following values:

- `true`: Use the default cache, as configured by the `kodo.DataCache` configuration property. This is the default when no extension is given, unless a superclass names a different cache.
- `false`: Data for instances of this class should not be cached.
- `<cache-name>`: Place data for instances of this class into the cache with name `<cache-name>`.

6.4.1.3. Detached State

The Kodo **enhancer** may add a synthetic field to detachable classes to hold detached state (see [Section 11.1.3, “Defining the Detached Object Graph” \[578\]](#) for details). You can instead declare your own detached state field or suppress the creation of a detached state field altogether. In the latter case, your class must not use **datastore identity**, and should declare a version field to detect optimistic concurrency errors during detached modifications.

Kodo JPA defines the `kodo.persistence.DetachedState` annotation for controlling detached state. When used to annotate a class, `DetachedState` recognizes the following properties:

- `boolean enabled`: Set to `false` to suppress the use of detached state.
- `String fieldName`: Use this property to declare your own detached state field. The field must be of type `Object`. Typically this property is only used if the field is inherited from a non-persisted superclass. If the field is declared in your entity class, you will typically annotate the field directly, as described below.

If you declare your own detached state field, you can annotate that field with `DetachedState` directly, rather than placing the annotation at the class level and using the `fieldName` property. When placed on a field, `DetachedState` acts as a marker annotation; it does not recognize any properties. Your annotated field must be of type `Object`.

The JDO `detached-state-field` extension key names the `Object` field used to store an object's detached state information. Set this extension to the name of a field in your class, or to `false` to disable the use of detached state.

6.4.1.4. Lock Groups

Kodo requires you to pre-declare subclass lock groups in the least-derived mapped class. The JPA `kodo.persistence.LockGroups` annotation accepts an array of lock group names. The JDO `lock-groups` XML extension key accepts a string of comma-separated lock group names. For details on lock groups, see [Section 5.8.1, “Lock Groups”](#)

and Subclasses” [477].

6.4.1.5. Auditable

Reserved for future use.

6.4.2. Field Extensions

Kodo recognizes the following field extensions:

6.4.2.1. Dependent

In a *dependent* relation, the referenced object is deleted whenever the owning object is deleted, or whenever the relation is severed by nulling or resetting the owning field. For example, if the `Magazine.coverArticle` field is marked dependent, then setting `Magazine.coverArticle` to a new `Article` instance will automatically delete the old `Article` stored in the field. Similarly, deleting a `Magazine` object will automatically delete its current `coverArticle`. You can prevent an orphaned dependent object from being deleted by assigning it to another relation in the same transaction.

JDO standardizes support for dependent relations with the `dependent` attribute of the XML `field` element, and the `dependent-element`, `dependent-key`, and `dependent-value` attributes of the `collection` and `map` elements. Kodo JPA offers a family of marker annotations to denote dependent relations in JPA entities:

- `kodo.persistence.Dependent`: Marks a direct relation as dependent.
- `kodo.persistence.ElementDependent`: Marks the entity elements of a collection, array, or map field as dependent.
- `kodo.persistence.KeyDependent`: Marks the key entities in a map field as dependent.

6.4.2.2. LRS

This boolean extension, denoted by the JPA `kodo.persistence.LRS` annotation, or the JDO `lrs` metadata extension key, indicates that a field should use Kodo's special large result set collection or map proxies. A complete description of large result set proxies is available in [Section 5.5.4.2, “Large Result Set Proxies” \[461\]](#).

6.4.2.3. Order-By

The JPA Overview's [Section 5.2.12, “Order By” \[41\]](#) describes JPA's `OrderBy` annotation for loading the elements of collection fields in a prescribed order. The JDO `order-by` metadata extension serves the same function. Ordering syntax is as follows:

```
#element|<field name>[ asc|ascending|desc|descending][, ...]
```

The token `#element` represents the element value. Simple element types such as strings and primitive wrappers are sorted based on their natural ordering. If the collection holds persistent objects, its elements are sorted based on the natural ordering of the objects' primary key values. By substituting a field name for the `#element` token, you can order a collection of persistent objects by an arbitrary field in the related type, rather than by primary key.

The field name or `#element` token may be followed by the keywords `asc/ascending` or `desc/descending` in either all-upper or all-lower case to mandate ascending and descending order. If the direction is omitted, Kodo defaults to ascending order.

Note that the defined ordering is only applied when the collection is loaded from the datastore. It is not maintained by Kodo as you modify the collection in memory.

The following ordering string orders a collection by its element values in descending order:

```
"#element desc"
```

The following ordering string orders a collection of `Author` objects by each author's last name in ascending order. If two last names are equal, the authors are ordered by first name in ascending order.

```
"firstName, lastName"
```

6.4.2.4. Inverse-Logical

This extension names the inverse field in a logical bidirectional relation. To create a logical bidirectional relation in Kodo JPA, use the `kodo.persistence.InverseLogical` annotation. To create a logical bidirectional relation in Kodo JDO, use the `inverse-logical` XML extension key. We discuss logical bidirectional relations and this extension in detail in [Section 5.4, “Managed Inverses” \[458\]](#).

6.4.2.5. Lock Group

Lock groups allow for fine-grained optimistic locking concurrency. Use Kodo JPA's `kodo.persistence.LockGroup` annotation or Kodo JDO's `lock-group` extension key to name the lock group for a field. You can exclude a field from optimistic locking with a value of `none`. We discuss lock groups and this extension further in [Section 5.8, “Lock Groups” \[476\]](#).

6.4.2.6. Read-Only

The read-only extension makes a field unwritable. The extension only applies to existing persistent objects; new object fields are always writeable.

To mark a field read-only in JPA metadata, set the `kodo.persistence.ReadOnly` annotation to a `kodo.persistence.UpdateAction` enum value. The `UpdateAction` enum includes:

- `UpdateAction.IGNORE`: Updates to the field are completely ignored. The field is not considered dirty. The new value will not even get stored in the Kodo **data cache**.
- `UpdateAction.RESTRICT`: Any attempt to change the field will result in an immediate exception.

To mark a field read-only in JDO metadata, set the `read-only` extension key to one of the following values:

- `ignore`: Updates to the field are completely ignored. The field is not considered dirty. The new value will not even get stored in the Kodo **data cache**.
- `restrict`: Any attempt to change the field will result in an immediate exception.

6.4.2.7. Type

Kodo has three levels of support for relations:

1. Relations that hold a reference to an object of a concrete persistent class are supported by storing the primary key values of the related instance in the database.
2. Relations that hold a reference to an object of an unknown persistent class are supported by storing the stringified identity value of the related instance. This level of support does not allow queries across the relation.
3. Relations that hold an unknown object or interface. The only way to support these relations is to serialize their value to the database. This does not allow you to query the field, and is not very efficient.

Clearly, when you declare a field's type to be another persistence-capable class, Kodo uses level 1 support. By default, Kodo assumes that any interface-typed fields you declare will be implemented only by other persistent classes, and assigns interfaces level 2 support. The exception to this rule is the `java.io.Serializable` interface. If you declare a field to be of type `Serializable`, Kodo lumps it together with `java.lang.Object` fields and other non-interface, unrecognized field types, which are all assigned level 3 support.

With Kodo's type family of metadata extensions, you can control the level of support given to your unknown/interface-typed fields. Setting the value of this extension to `Entity` in JPA or `PersistenceCapable` in JDO indicates that the field value will always be some persistent object, and gives level 2 support. Setting the value of this extension to the class of a concrete persistent type is even better; it gives you level 1 support (just as if you had declared your field to be of that type in the first place). Setting this extension to `Object` uses level 3 support. This is useful when you have an interface relation that may **not** hold other persistent objects (recall that Kodo assumes interface fields will always hold persistent instances by default).

This extension is also used with Kodo's externalization feature, described in [Section 5.5.5, “Externalization” \[464\]](#).

Kodo JPA defines the following type annotations for field values, collection, array, and map elements, and map keys, respectively:

- `kodo.persistence.Type`
- `kodo.persistence.ElementType`
- `kodo.persistence.KeyType`

Kodo JDO defines the following type extension keys for field values, collection and array elements, map entry keys, and map entry values, respectively:

- `type`
- `element-type`
- `key-type`
- `value-type`

6.4.2.8. Externalizer

The JPA `kodo.persistence.Externalizer` annotation or JDO `externalizer` extension key names a method to transform a field value into a value of another type. See [Section 5.5.5, “Externalization” \[464\]](#) for details.

6.4.2.9. Factory

The JPA `kodo.persistence.Factory` annotation or JDO `factory` extension key names a method to re-create a field value from its externalized form. See [Section 5.5.5, “Externalization” \[464\]](#) for details.

6.4.2.10. External Values

The JPA `kodo.persistence.ExternalValues` annotation or JDO `external-values` extension key declares values for transformation of simple fields to different constant values in the datastore. See [Section 5.5.5.1, “External Values” \[468\]](#) for details.

6.4.3. Example

The following example shows you how to specify extensions in metadata.

Example 6.4. Kodo Metadata Extensions

JPA:

```
import kodo.persistence.*;

@Entity
@DataCache(enabled=false)
public class Magazine
{
    @ManyToMany
    @LRS
    @LockGroup(LockGroup.NONE)
    private Collection<Subscriber> subscribers;

    @ExternalValues({"true=1", "false=2"})
    @Type(int.class)
    private boolean weekly;

    @PersistentCollection
    @OrderBy("#element DESC")
    private List<String> subtitles;

    ...
}
```

JDO:

```
<jdo>
  <package name="org.mag">
    <class name="Magazine">
      <extension vendor-name="kodo" key="data-cache" value="false"/>
      <field name="subscribers">
        <collection element-type="Subscriber"/>
        <extension vendor-name="kodo" key="lrs" value="true"/>
        <extension vendor-name="kodo" key="lock-group" value="none"/>
      </field>
      <field name="weekly">
        <extension vendor-name="kodo" key="external-values"
          value="true=1,false=2"/>
        <extension vendor-name="kodo" key="type" value="int"/>
      </field>
      <field name="subtitles">
        <collection element-type="string"/>
        <extension vendor-name="kodo" key="order-by"
          value="#element desc"/>
      </field>
    </class>
  </package>
</jdo>
```

Chapter 7. Mapping

The JPA Overview's [Chapter 12, Mapping Metadata](#) [118] explains mapping under JPA. The JDO Overview's [Chapter 15, Mapping Metadata](#) [260] explains object-relational mapping under JDO. This chapter reviews the mapping utilities Kodo provides and examines Kodo features that go beyond the specifications.

7.1. Forward Mapping

Forward mapping is the process of creating mappings and their corresponding database schema from your object model. Kodo supports forward mapping through the *mapping tool*. The next section presents several common mapping tool use cases. You can invoke the tool through the `mappingtool` shell/batch script included in the Kodo distribution, or through its Java class, `kodo.jdbc.meta.MappingTool`.

Note

[Section 14.1.4, “Mapping Tool Ant Task”](#) [610] describes the mapping tool Ant task.

Example 7.1. Using the Mapping Tool

```
mappingtool Magazine.java
```

```
mappingtool *.jdo
```

In addition to the universal flags of the **configuration framework**, the mapping tool accepts the following command line arguments:

- `-file/-f <stdout | output file>`: Use this option to write the planned mappings to an XML document rather than recording them as the mappings for the given classes. This option also specifies the metadata file to write to when using the mapping tool to generate default persistence metadata (see [Section 6.1, “Generating Default JDO Metadata”](#) [480]), or the file to dump to if using the `export` action.
- `-schemaAction/-sa <add | refresh | drop | build | retain | none>`: The action to take on the schema. These options correspond to the same-named actions on the schema tool described in [Section 4.13, “Schema Tool”](#) [444]. Unless you are running the mapping tool on all of your persistent types at once or dropping a mapping, we strongly recommend you use the default `add` action or the `build` action. Otherwise you may end up inadvertently dropping schema components that are used by classes you are not currently running the tool over.
- `-schemaFile/-sf <stdout | output file>`: Use this option to write the planned schema to an XML document rather than modify the database. The document can then be manipulated and committed to the database with the **schema tool**.
- `-sqlFile/-sql <stdout | output file>`: Use this option to write the planned schema modifications to a SQL script rather than modify the database. Combine this with a `schemaAction` of `build` to generate a script that recreates the schema for the current mappings, even if the schema already exists.
- `-dropTables/-dt <true/t | false/f>`: Corresponds to the same-named option on the schema tool.

- `-dropSequences/-dsq <true/t | false/f>`: Corresponds to the same-named option on the schema tool.
- `-kodoTables/-kt <true/t | false/f>`: Corresponds to the same-named option on the schema tool.
- `-ignoreErrors/-i <true/t | false/f>`: Corresponds to the same-named option on the schema tool.
- `-schemas/-s <schema and table names>`: Corresponds to the same-named option on the schema tool. This option is ignored if `readSchema` is not set to `true`.
- `-readSchema/-rs <true/t | false/f>`: Set this option to `true` to read the entire existing schema when the tool runs. Reading the existing schema ensures that Kodo does not generate any mappings that use table, index, primary key, or foreign key names that conflict with existing names. Depending on the JDBC driver, though, it can be a slow process for large schemas.
- `-primaryKeys/-pk <true/t | false/f>`: Whether to read and manipulate primary key information of existing tables. Defaults to `false`.
- `-foreignKeys/-fk <true/t | false/f>`: Whether to read and manipulate foreign key information of existing tables. Defaults to `false`. This means that to add any new foreign keys to a class that has already been mapped, you must explicitly set this flag to `true`.
- `-indexes/-ix <true/t | false/f>`: Whether to read and manipulate index information of existing tables. Defaults to `false`. This means that to add any new indexes to a class that has already been mapped once, you must explicitly set this flag to `true`.
- `-sequences/-sq <true/t | false/f>`: Whether to manipulate sequences. Defaults to `true`.
- `-meta/-m <true/t | false/f>`: Whether the given action applies to metadata rather than or in addition to mappings.

The mapping tool also uses an `-action/-a` argument to specify the action to take on each class. The available actions are:

- `refresh`: This is the default action when using JDO mapping defaults (see [Section 7.4, “Mapping Defaults” \[500\]](#)) It brings your mappings up-to-date with the class definitions. Kodo will attempt to use any provided mapping information, and fill in missing information. If the provided mappings conflict with a class definition, the conflicting mappings will be discarded and the class or field will be re-mapped to a new column or table.
- `add`: If used with the `-meta` option, adds new default metadata for the given classes. Otherwise, brings your mappings up-to-date with the class definitions. Kodo will attempt to use any provided mapping information, and fill in missing information. If the provided mappings conflict with a class definition, Kodo will fail with an informative exception.
- `buildSchema`: This is the default action when using JPA mapping defaults (see [Section 7.4, “Mapping Defaults” \[500\]](#)) It makes the database schema match your existing mappings. If your provided mappings conflict with a class definition, Kodo will fail with an informative exception.
- `drop`: Delete the mappings for the given classes. If used with the `-meta` option, also deletes persistence metadata
- `validate`: Ensure that the mappings for the given classes are valid and that they match the schema. No mappings or tables will be changed. An exception is thrown if any mappings are invalid.
- `import`: Import mappings from the given XML document and store them as the current system mappings. Under this action, the mapping tool expects its arguments to be XML files in the `orm` mapping format described in [Chapter 15, Mapping Metadata \[260\]](#) of the JDO Overview, rather than persistent classes.
- `export`: Export the mapping data for the given classes to an XML file in the `orm` mapping format described in [Chapter 15, Mapping Metadata \[260\]](#) of the JDO Overview.

Note

When using JPA annotation mappings, you cannot run the `refresh`, `add`, or `drop` mapping tool actions. Each of these actions changes existing mappings, and Kodo cannot yet write your annotations for you.

Each additional argument to the tool should be one of:

- The full name of a persistent class.
- The `.java` file for a persistent class.
- The `.class` file of a persistent class.
- A `.jdo` metadata file. The tool will run on each class listed in the metadata.
- A `.orm` mapping file. The tool will run on each class listed in the mapping metadata. If you are running the `import` action, each argument *must* be a `.orm` file.

If you do not supply any arguments to the mapping tool, it will run on the classes in your persistent classes list (see [Section 5.1, “Persistent Class List” \[452\]](#)).

If you have not specified a persistent class list, the tool will scan your classpath for directories containing `.jdo` files, and run on all classes listed in those files.

The mappings generated by the mapping tool are stored by the system *mapping factory*. [Section 7.5, “Mapping Factory” \[503\]](#) discusses your mapping factory options.

7.1.1. Using the Mapping Tool

The JPA specification defines a comprehensive set of defaults for missing mapping information. Thus, forward mapping in JPA is virtually automatic. After using the mapping annotations covered in [Chapter 12, *Mapping Metadata* \[118\]](#) of the JPA Overview to override any unsatisfactory defaults, run the mapping tool's `buildSchema` action on your persistent classes. This is the default action when you use JPA mapping defaults (see [Section 7.4, “Mapping Defaults” \[500\]](#)).

The `buildSchema` action manipulates the database schema to match your mappings. It fails if any of your mappings don't match your object model.

Example 7.2. Creating the Relational Schema from Mappings

JPA mapping defaults:

```
mappingtool Magazine.java
```

JDO mapping defaults:

```
mappingtool -a buildSchema package.jdo
```

In standard forward mapping under JDO, you concentrate your efforts on your object model, and the mapping tool's `refresh` action keeps your mappings and schema up-to-date. The refresh action examines both the existing database schema and any existing mappings. Classes and fields that are not mapped, or whose mapping information no longer matches the object model, are automatically given new mappings. The tool also updates the schema as necessary to support both existing mappings and any new mappings it creates. The example below shows how to invoke the refresh action on the mapping tool to create or update the mapping information and database schema for the persistent classes listed in `package.jdo`.

Example 7.3. Refreshing Mappings and the Relational Schema

```
mappingtool package.jdo
```

You can safely run the `refresh` action on classes that have already been mapped, because the tool only generates new mappings when the old ones have become incompatible with the class. If the tool does have to replace a bad mapping, it does not modify other still-valid mappings. For example, if you change the type of a field from `int` to `String`, the mapping tool will detect the incompatibility with the old numeric column, add a new string-compatible column to the class' database table, and change the field's mapping data to point to the new column. All other fields will retain their original mappings.

In fact, if you want to make sure the mapping tool does not alter any of your existing mappings, you can use the `add` action in place of the default `refresh` action. When the mapping tool encounters what it thinks is a bad mapping under the `add` action, it throws an informative exception rather than replacing the mapping. This is particularly useful if you write mappings by hand, but want the tool to create the corresponding schema for you.

Example 7.4. Adding Mappings and the Relational Schema

```
mappingtool -a add package.jdo
```

To drop JDO mapping data, use the `drop` action. This action does not affect the schema.

Example 7.5. Dropping Mappings

```
mappingtool -a drop package.jdo
```

To drop the schema for a persistent class, set the mapping tool's `schemaAction` to `drop`.

Example 7.6. Dropping Mappings and Association Schema

```
mappingtool -sa drop Magazine.java
```

```
mappingtool -sa drop package.jdo
```

7.1.2. Generating DDL SQL

The examples below show how to use the mapping tool to generate DDL SQL scripts, rather than modifying the database directly.

Example 7.7. Create DDL for Current Mappings

This example uses your existing mappings to determine the needed schema, then writes the SQL to create that schema to `create.sql`.

```
mappingtool -a buildSchema -sa build -sql create.sql Magazine.java
```

Example 7.8. Create DDL to Update Database for Current Mappings

This example uses your existing mappings to determine the needed schema. It then writes the SQL to add any missing tables and columns to the current schema to `update.sql`.

```
mappingtool -a buildSchema -sql update.sql Magazine.java
```

Example 7.9. Refresh JDO Mappings and Create DDL

This example refreshes the mappings for the classes in `package.jdo` and writes all the SQL necessary to recreate the tables used by these mappings to `create.sql`.

```
mappingtool -sa build -sql create.sql package.jdo
```

Example 7.10. Refresh JDO Mappings and Create DDL to Update Database

This example refreshes the mappings for the classes in `package.jdo`. It writes the SQL to add tables or columns missing from the current schema to the `update.sql` file.

```
mappingtool -sql update.sql package.jdo
```

7.1.3. JDO Forward Mapping Hints

Forward mapping in Kodo JDO is not an all-or-nothing endeavor. Kodo allows you to specify bits and pieces of mapping information; the mapping tool will fill in the rest. For example, if you want the column for your `Magazine.isbn` field to be type `CHAR(15)`, or you want to use the `new-table` inheritance strategy to create a joined subclass table, you can specify these pieces of information without filling in column names, table names, and other data whose defaults are satisfactory.

Example 7.11. Partial Mapping

```
<class name="Magazine">
  <field name="isbn">
    <column jdbc-type="char" length="15"/>
  </field>
  ...
</class>
<class name="FullTimeEmployee"> <!-- extends Employee -->
  <inheritance strategy="new-table"><join/></inheritance>
  ...
</class>
```

Important

In the `FullTimeEmployee` example above, the `join` within the `inheritance` element is very important. Recall that the `join` element is all that separates a joined inheritance mapping from a table-per-class mapping. Thus, specifying a `new-table` subclass strategy without using a `join` element hint will cause Kodo to create an unjoined table-per-class mapping instead of a joined mapping.

The mapping tool will incorporate your partial mapping hints into the full mappings it creates. The tool even allows you to specify the hints in your `.jdo` file when your mappings are stored somewhere else, such as separate `.orm` files. This allows you to integrate mapping hints into the metadata for unmapped classes or newly-added fields without having to write a separate mapping file. This also brings up a potential conflict, however: when you have mapping information in both your `.jdo` file and the configured mapping format such as a `.orm` file, which takes precedence during tool runs? The answer is that the data in the configured mapping format always wins, just as it does at runtime. Mapping hints only work for classes or fields that have not yet been mapped. If you have configured Kodo to store mappings in `.orm` files, the mapping tool will ignore mapping hints for any class or field already mentioned in your `.orm` mapping information.

In addition to hinting through partial mapping, the mapping tool recognizes some general directives communicated through mapping metadata:

- When mapping an embedded relation, you can set the embedded element's `null-indicator-column` attribute to `true` to have Kodo create a synthetic null indicator column with a default name. You can also set this attribute to the name of a field in the embedded class. Kodo will use that field's column as the null indicator column. Of course, Kodo also allows you to specify a column name directly, as the attribute intends.
- You can set the `column` attribute of the `order` element to `true` or `false` to force or suppress the creation of an ordering column for a collection field.

7.1.4. Runtime Forward Mapping

You can configure Kodo to automatically run the mapping tool at runtime through the `kodo.jdbc.SynchronizeMappings` configuration property. Using this property saves you the trouble of running the mapping tool manually, and is meant for use during rapid test/debug cycles.

In order to enable automatic runtime mapping, you must first list all your persistent classes as described in [Section 5.1, “Persistent Class List” \[452\]](#).

Kodo will run the mapping tool on these classes when your application obtains its first `EntityManager` or `PersistenceManager`.

The `kodo.jdbc.SynchronizeMappings` property is a plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) where the class name is the mapping tool action to invoke, and the properties are the `MappingTool` class' `JavaBean` properties. These properties correspond to the long versions of the tool's command line flags.

Example 7.12. Configuring Runtime Forward Mapping

JPA XML format:

```
<property name="kodo.jdbc.SynchronizeMappings" value="buildSchema(ForeignKeys=true)" />
```

JDO properties format:

```
kodo.jdbc.SynchronizeMappings: buildSchema(ForeignKeys=true)
```

The setting above corresponds to running the following command:

```
mappingtool -a buildSchema -fk true
```

7.2. Reverse Mapping

Kodo includes a *reverse mapping* tool for generating persistent class definitions, complete with metadata, from an existing database schema. You do not have to use the reverse mapping tool to access an existing schema; you are free to write your classes and mappings yourself, as described in [Section 7.3, “Meet-in-the-Middle Mapping” \[499\]](#). The reverse mapping tool, however, can give you an excellent starting point from which to grow your persistent classes.

Note

The reverse mapping tool in this release does not yet output JPA annotations. It is limited to JDO metadata output.

To use the reverse mapping tool, follow the steps below:

1. Use the **schema tool** to export your current schema to an XML schema file. You can skip this step and the next step if you want to run the reverse mapping tool directly against the database.

Example 7.13. Reflection with the Schema Tool

```
schematool -a reflect -f schema.xml
```

2. Examine the generated schema file. JDBC drivers often provide incomplete or faulty metadata, in which case the file will not exactly match the actual schema. Alter the XML file to match the true schema. The XML format for the schema file is described in [Section 4.14, “XML Schema Format” \[447\]](#).

After fixing any errors in the schema file, modify the XML to include foreign keys between all relations. The schema tool will have automatically detected existing foreign key constraints; many schemas, however, do not employ database foreign keys for every relation. By manually adding any missing foreign keys, you will give the reverse mapping tool the information it needs to generate the proper relations between the persistent classes it creates.

3. Run the reverse mapping tool on the finished schema file. If you do not supply the schema file to reverse map, the tool will run directly against the schema in the database. The tool can be run via the included `reversemappingtool` script, or through its Java class, `kodo.jdbc.meta.ReverseMappingTool`.

Example 7.14. Using the Reverse Mapping Tool

```
reversemappingtool -pkg com.xyz -d ~/src -cp customizer.properties schema.xml
```

In addition to Kodo's **standard configuration flags**, including **code formatting options**, the reverse mapping tool recognizes the following command line arguments:

- `-schemas/-s <schema and table names>` : A comma-separated list of schema and table names to reverse map, if no XML schema file is supplied. Each element of the list must follow the naming conventions for the `kodo.jdbc.Schemas` property described in [Section 4.12.1, “Schemas List” \[443\]](#). In fact, if this flag is omitted, it defaults to the value of the `Schemas` property. If the `Schemas` property is not defined, all schemas will be reverse-mapped.
- `-package/-pkg <package name>`: The package name of the generated classes. If no package name is given, the generated code will not contain package declarations.
- `-directory/-d <output directory>` : All generated code and metadata will be written to the directory at this path. If the path does not match the package of a class, the package structure will be created beneath this directory. Defaults to the current directory.
- `-useSchemaName/-sn <true/t | false/f>` : Set this flag to `true` to include the schema as well as table name in the name of each generated class. This can be useful when dealing with multiple schemas with same-named tables.
- `-useForeignKeyName/-fkn <true/t | false/f>`: Set this flag to `true` if you would like field names for relations to be based on the database foreign key name. By default, relation field names are derived from the name of the related class.

- `-nullableAsObject/-no <true/t | false/f>` : By default, all non-foreign key columns are mapped to primitives. Set this flag to `true` to generate primitive wrapper fields instead for columns that allow null values.
- `-blobAsObject/-bo <true/t | false/f>` : By default, all binary columns are mapped to `byte[]` fields. Set this flag to `true` to map them to `Object` fields instead. Note that when mapped this way, the column is presumed to contain a serialized Java object.
- `-primaryKeyOnJoin/-pkj <true/t | false/f>` : The standard reverse mapping tool behavior is to map all tables with primary keys to persistent classes. If your schema has primary keys on many-many join tables as well, set this flag to `true` to avoid creating classes for those tables.
- `-inverseRelations/-ir <true/t | false/f>`: Set to `false` to prevent the creation of inverse 1-many/1-1 relations for every many-1/1-1 relation detected.
- `-useDatastoreIdentity/-ds <true/t | false/f>`: Set to `true` to use datastore identity for tables that have single numeric primary key columns. The tool typically uses application identity for all generated classes.
- `-useBuiltinIdentityClass/-bic <true/t | false/f>`: Set to `false` to prevent the tool from using built-in application identity classes when possible. This will force the tool to create custom application identity classes even when there is only one primary key column.
- `-innerIdentityClasses/-inn <true/t | false/f>`: Set to `true` to have any generated application identity classes be created as static inner classes within the persistent classes. Defaults to `false`.
- `-identityClassSuffix/-is <suffix>`: Suffix to append to class names to form application identity class names, or for inner identity classes, the inner class name. Defaults to `Id`.
- `-discriminatorStrategy/-ds <strategy>`: The default discriminator strategy to use for generated base classes. Defaults to a strategy using outer joins to all possible subclass tables to determine a record's type. This strategy does not require a discriminator column.
- `-versionStrategy/-vs <strategy>`: The default version strategy to use for generated base classes. Defaults to using state comparisons on commit to detect concurrency violations.
- `-detachable/-det <true/t | false/f>`: Whether to declare the generated classes detachable in metadata. Defaults to `false` for JDO classes.
- `-metadata/-md <package | class>`: Whether to write a single package-level metadata file, or to write a metadata file per generated class. Defaults to `package`.
- `-typeMap/-typ <type mapping>`: A string that specifies the default Java classes to generate for each SQL type that is seen in the schema. The format is `SQLTYPE1=JavaClass1,SQLTYPE2=JavaClass2`. The SQL type name first looks for a customization based on `SQLTYPE(SIZE,PRECISION)`, then `SQLTYPE(SIZE)`, then `SQLTYPE(SIZE,PRECISION)`. So if a column whose type name is `CHAR` is found, it will first look for the `CHAR(50,0)` type name specification, then it will look for `CHAR(50)`, and finally it will just look for `CHAR`. For example, to generate a char array for every `CHAR` column whose size is exactly 50, and to generate a `short` for every type name of `INTEGER`, you might specify: `CHAR(50)=char[],INTEGER=short`. Note that since various databases report different type names differently, one database's type name specification might not work for another database. Enable `TRACE` level logging on the `MetaData` channel to track which type names Kodo is examining.
- `-customizerClass/-cc <class name>`: The full class name of a `kodo.jdbc.meta.ReverseCustomizer` customization plugin. If you do not specify a reverse customizer of your own, the system defaults to a `PropertiesReverseCustomizer`. This customizer allows you to specify simple customization options in the properties file given with the `-customizerProperties` flag below. We present the available property keys **below**.
- `-customizerProperties/-cp <properties file or resource>`: The path or resource name of a properties file to pass to the reverse customizer on initialization.

- `-customizer./-c.<property name> <property value>`: The given property name will be matched with the corresponding Java bean property in the specified reverse customizer, and set to the given value.

Running the tool will generate `.java` files for each generated class (and its application identity class, if applicable), along with all necessary persistence metadata and mappings.

4. Examine the generated class, metadata, and mapping information, and modify it as necessary. Remember that the reverse mapping tool only provides a starting point, and you are free to make whatever modifications you like to the code it generates.

After you are satisfied with the generated classes and their mappings, you should first compile them with `javac`, `jikes`, or your favorite Java compiler. Make sure the classes and their metadata are located in the directory corresponding to the `-package` flag you gave the reverse mapping tool. Finally, enhance the classes if necessary (see [Section 5.2](#), “**Enhancement**” [452]).

Your persistent classes are now ready to access your existing schema.

7.2.1. Customizing Reverse Mapping

The `kodo.jdbc.meta.ReverseCustomizer` plugin interface allows you to customize the reverse mapping process. See the class [Javadoc](#) for details on the hooks that this interface provides. Specify the concrete plugin implementation to use with the `-customizerClass/-cc` command-line flag, described in the preceding section.

By default, the reverse mapping tool uses a `kodo.jdbc.meta.PropertiesReverseCustomizer`. This customizer allows you to perform relatively simple customizations through the properties file named with the `-customizerProperties` tool flag. The customizer recognizes the following properties:

- `<table name>.table-type <type>` : Override the default type of the table with name `<table name>`. Legal values are:
 - `base`: Primary table for a base class.
 - `secondary`: Secondary table for a class. The table must have a foreign key joining to a class table.
 - `secondary-outer`: Outer-joined secondary table for a class. The table must have a foreign key joining to a class table.
 - `association`: Association table. The table must have two foreign keys to class tables.
 - `collection`: Collection table. The table must have one foreign key to a class table and one data column.
 - `subclass`: A joined subclass table. The table must have a foreign key to the superclass' table.
 - `none`: The table should not be reverse-mapped.
- `<class name>.rename <new class name>` : Override the given tool-generated name `<class name>` with a new value. Use full class names, including package. You are free to rename a class to a new package. Specify a value of `none` to reject the class and leave the corresponding table unmapped.
- `<table name>.class-name <new class name>`: Assign the given fully-qualified class name to the type created from the table with name `<table name>`. Use a value of `none` to prevent reverse mapping this table. This property can be used in place of the `rename` property.
- `<class name>.identity <datastore | builtin | identity class name>`: Set this property to `datastore` to use datastore identity for the class `<class name>`, `builtin` to use a built-in identity class, or the desired application identity class name. Give full class names, including package. You are free to change the package of the identity class this way. If the persistent class has been renamed, use the new class name for this property key. Remember that data-

store identity requires a table with a single numeric primary key column, and built-in identity requires a single primary key column of any type.

- `<class name>.<field name>.rename <new field name>`: Override the tool-generated `<field name>` in class `<class name>` with the given name. Use the field owner's full class name in the property key. If the field owner's class was renamed, use the new class name. The property value should be the new field name, without the preceding class name. Use a value of `none` to reject the generated mapping and remove the field from the class.
- `<table name>.<column name>.field-name <new field name>`: Set the generated field name for the `<table name>` table's `<column name>` column. If this is a multi-column mapping, any of the columns can be used. Use a value of `none` to prevent the column and its associated columns from being reverse-mapped.
- `<class name>.<field name>.type <field type>`: The type to give the named field. Use full class names. If the field or the field's owner class has been renamed, use the new name.
- `<class name>.<field name>.value` : The initial value for the named field. The given string will be placed as-is in the generated Java code, so be sure it is valid Java. If the field or the field's owner class has been renamed, use the new name.

All property keys are optional; if not specified, the customizer keeps the default value generated by the reverse mapping tool.

Example 7.15. Customizing Reverse Mapping with Properties

```
reversemappingtool -pkg com.xyz -cp custom.properties schema.xml
```

Example `custom.properties`:

```
com.xyz.TblMagazine.rename:      com.xyz.Magazine
com.xyz.TblArticle.rename:      com.xyz.Article
com.xyz.TblPubCompany.rename:   com.xyz.pub.Company
com.xyz.TblSysInfo.rename:      none

com.xyz.Magazine.allArticles.rename:  articles
com.xyz.Magazine.articles.type:      java.util.Collection
com.xyz.Magazine.articles.value:     new TreeSet()
com.xyz.Magazine.identity:         datastore

com.xyz.pub.Company.identity:       com.xyz.pub.CompanyId
```

Your Kodo download includes the `PropertiesReverseCustomizer` source code. You can use this code as an example when writing your own customization class.

7.3. Meet-in-the-Middle Mapping

In the *meet-in-the-middle* mapping approach, you control both the relational model and the object model. It is up to you to define the mappings between these models. Kodo has two tools to aid meet-in-the-middle mapping. First, the Kodo Workbench includes a visual interface to wiring your classes and schema together. Second, the mapping tool's `validate` action is useful to meet-in-the-middle mappers. We examined the mapping tool in [Section 7.1, “Forward Mapping” \[489\]](#). The `validate` action verifies that the mapping information for a class matches the class definition and the existing schema. It throws an informative exception when your mappings are incorrect.

Example 7.16. Validating Mappings

```
mappingtool -a validate Magazine.java
```

```
mappingtool -a validate package.jdo
```

The `buildSchema` action we discussed in [Section 7.1, “Forward Mapping” \[489\]](#) is also somewhat useful during meet-in-the-middle mapping. Unlike the `validate` action, which throws an exception if your mapping data does not match the existing schema, the `buildSchema` action assumes your mapping data is correct, and modifies the schema to match your mappings. This lets you modify your mapping data manually, but saves you the hassle of using your database's tools to bring the schema up-to-date.

`buildSchema` is the default action when you use JPA mapping defaults ([Section 7.4, “Mapping Defaults” \[500\]](#))

Example 7.17. Creating the Relational Schema from Mappings

JPA mapping defaults:

```
mappingtool Magazine.java
```

JDO mapping defaults:

```
mappingtool -a buildSchema package.jdo
```

7.4. Mapping Defaults

The previous sections showed how to use the mapping tool to generate default mappings. But how does the mapping tool know what mappings to generate? The answer lies in the `kodo.jdbc.meta.MappingDefaults` interface. Kodo uses an instance of this interface to decide how to name tables and columns, where to put foreign keys, and generally how to create a schema that matches your object model.

Important

Kodo relies on foreign key constraint information at runtime to order SQL appropriately. Be sure to set your mapping defaults to reflect your existing database constraints, or use explicit foreign key mappings as described in [Section 7.7.9.2, “Foreign Keys” \[517\]](#) for JPA, and [Section 15.12, “Foreign Keys” \[309\]](#) of the JDO Overview .

The `kodo.jdbc.MappingDefaults` configuration property controls the `MappingDefaults` interface implementation in use. This is a plugin property (see [Section 2.5, “Plugin Configuration” \[395\]](#)), so you can substitute your own implementation or configure the existing ones. Kodo includes the following standard implementations:

- `jpa`: Provides defaults in compliance with the JPA standard. This is an alias for the `kodo.persistence.jdbc.PersistenceMappingDefaults` class. This class extends the `MappingDefaultsImpl` class described below, so it has all the same properties (though with different default values).
- `jdo`: This is an alias for the `kodo.jdbc.meta.MappingDefaultsImpl` class. This default implementation is highly configurable. It has the following properties:
 - `DefaultMissingInfo`: Whether to default missing column and table names rather than throw an exception. Defaults to false, meaning full mappings are required at runtime and when using mapping tool actions like `buildSchema` and `validate`.

When this property is false and you use mapping tool actions like `refresh` or `add` to create new mappings, Kodo ensures that the table and column names it generates do not conflict among mappings. If you have set this property to true, however, Kodo does not attempt to avoid conflicts, because doing so would make mapping non-deterministic.

The `jpa` plugin above sets this property to true to meet the JPA specification.

- `BaseClassStrategy`: The default mapping strategy for base classes. You can specify a built-in strategy alias or the full class name of a **custom class strategy**. You can also use Kodo's plugin format (see [Section 2.5, “Plugin Configuration” \[395\]](#)) to pass arguments to the strategy instance. See the `kodo.jdbc.meta.strats` package for available strategies.
- `SubclassStrategy`: The default mapping strategy for subclasses. You can specify a builtin strategy alias or the full class name of a **custom class strategy**. You can also use Kodo's plugin format (see [Section 2.5, “Plugin Configuration” \[395\]](#)) to pass arguments to the strategy instance. Common strategies are `vertical` and `flat`, the default. See the `kodo.jdbc.meta.strats` package for all available strategies.
- `VersionStrategy`: The default version strategy for classes without a version field. You can specify a builtin strategy alias or the full class name of a **custom version strategy**. You can also use Kodo's plugin format (see [Section 2.5, “Plugin Configuration” \[395\]](#)) to pass arguments to the strategy instance. Common strategies are `none`, `state-comparison`, `timestamp`, and `version-number`, the default. See the `kodo.jdbc.meta.strats` package for all available strategies.
- `DiscriminatorStrategy`: The default discriminator strategy when no discriminator value is given. You can specify a builtin strategy alias or the full class name of a **custom discriminator strategy**. You can also use Kodo's plugin format (see [Section 2.5, “Plugin Configuration” \[395\]](#)) to pass arguments to the strategy instance. Common strategies are `final` for a base class without subclasses, `none` to use joins to subclass tables rather than a discriminator column, and `class-name`, the default. See the `kodo.jdbc.meta.strats` package for all available strategies.
- `FieldStrategies`: This property associates field types with custom strategies. The format of this property is similar to that of plugin strings (see [Section 2.5, “Plugin Configuration” \[395\]](#)), without the class name. It is a comma-separated list of key/value pairs, where each key is a possible field type, and each value is itself a plugin string describing the strategy for that type. We present an example below. See [Section 7.10.3, “Custom Field Mapping” \[523\]](#) for information on custom field strategies.
- `ForeignKeyDeleteAction`: The default delete action of foreign keys representing relations to other objects. Recognized values include `restrict`, `cascade`, `null`, `default`. These values correspond exactly to the standard database foreign key actions of the same names.

The value `none` tells Kodo not to create database foreign keys on relation columns. This is the default.

- `JoinForeignKeyDeleteAction`: The default delete action of foreign keys that join join secondary, collection, map, or subclass tables to the primary table. Accepts the same values as the `ForeignKeyDeleteAction` property above.

- `DeferConstraints`: Whether to use deferred database constraints if possible. Defaults to false.
- `IndexLogicalForeignKeys`: Boolean property controlling whether to create indexes on logical foreign keys. Logical foreign keys are columns that represent a link between tables, but have been configured through the `ForeignKey` properties above not to use a physical database foreign key. Defaults to true.
- `DataStoreIdColumnName`: The default name of datastore identity columns.
- `DiscriminatorColumnName`: The default name of discriminator columns.
- `IndexDiscriminator`: Whether to index the discriminator column. Defaults to true.
- `VersionColumnName`: The default name of version columns. If you use custom lock groups, this name may be combined with lock group names. See [Section 5.8, “Lock Groups” \[476\]](#) for more information on lock groups.
- `IndexVersion`: Whether to index the version column. Defaults to false.
- `AddNullIndicator`: Whether to create a synthetic null indicator column for embedded mappings. The null indicator column allows Kodo to distinguish between a null embedded object and one with default values for all persistent fields.
- `NullIndicatorColumnName`: The default name of synthetic null indicator columns for embedded objects.
- `OrderLists`: Whether to create a database ordering column for maintaining the order of persistent lists and arrays. Defaults to true.

The `jpa` plugin above sets this property to false in accordance with the JPA specification.

- `OrderColumnName`: The default name of collection and array ordering columns.
- `StoreEnumOrdinal`: Set to true to store enum fields as numeric ordinal values in the database. The default is to store the enum value name as a string, which is more robust if the Java enum declaration might be rearranged.
- `StoreUnmappedObjectIdString`: Set to true to store the stringified identity of related objects when the declared related type is unmapped. By default, Kodo stores the related object's primary key value(s). However, this breaks down if different subclasses of the related type use incompatible primary key structures. In that case, stringifying the identity value is the better choice.

The example below turns on foreign key generation during schema creation and associates the `org.mag.data.InfoStruct` field type with the custom `org.mag.mapping.InfoStructHandler` value handler.

Example 7.18. Configuring Mapping Defaults

JPA XML format:

```
<property name="kodo.jdbc.MappingDefaults"
  value="ForeignKeyDeleteAction=restrict, \
  FieldStrategies='org.mag.data.InfoStruct=org.mag.mapping.InfoStructHandler' "/>
```

JDO properties format:

```
kodo.jdbc.MappingDefaults: ForeignKeyDeleteAction=restrict, \
  FieldStrategies='org.mag.data.InfoStruct=org.mag.mapping.InfoStructHandler'
```

7.5. Mapping Factory

An important decision in the object-relational mapping process is how and where to store the data necessary to map your persistent classes to the database schema.

In JPA, mapping metadata is defined in annotations. Future versions of the JPA drafts will also define a mapping XML format. [Chapter 12, *Mapping Metadata* \[118\]](#) in the JPA Overview describes JPA mapping options.

In JDO, mapping metadata is integrated with persistence metadata in your `.jdo` files by default. [Section 15.1, “Mapping Metadata Placement” \[260\]](#) in the JDO Overview explains how to use separate `.orm` files for mapping metadata instead. Using separate files separates concerns and allows you to define multiple mappings for the same object model, but it is slightly less convenient to work with.

[Section 6.2, “Metadata Factory” \[480\]](#) introduced Kodo's `MetaDataFactory` interface. Kodo uses this same interface to abstract the storage and retrieval of mapping information. Kodo includes the built-in mapping factories below, and you can create your own factory if you have custom needs. You control which mapping factory Kodo uses with the `kodo.jdbc.MappingFactory` configuration property.

Kodo allows you to mix metadata and mapping factories from both the JPA and JDO specifications. For example, you can configure Kodo to use JPA annotations for metadata but JDO `.orm` files for mapping information. We present an example of this configuration below.

The bundled mapping factories are:

- `-`: Leaving the `kodo.jdbc.MappingFactory` property unset allows your metadata factory to take over mappings as well.

If you are using the `jpa` metadata factory, Kodo will read mapping information from your annotations when you leave the mapping factory unspecified.

If you are using the `jdo` metadata factory, this gives the standard behavior defined in [Section 15.1, “Mapping Metadata Placement” \[260\]](#). Even without setting an explicit mapping factory name, however, you can still use the `kodo.jdbc.MappingFactory` property to specify mapping options, including:

- `ConstraintNames`: Set this option to `true` to include the names of foreign key and unique constraints in all generated mappings. By default, Kodo does not record constraint names.
- `jdo-orm`: This is an alias for the `kodo.jdo.jdbc.ORMFileJDORMMappingFactory`. This factory stores mapping metadata in `.orm` files. It accepts the following properties:
 - `ConstraintNames`: Set this option to `true` to include the names of foreign key and unique constraints in all generated mappings. By default, Kodo does not record constraint names.
 - `Mapping`: The logical name of these mappings. Mapping files are suffixed with `"-<logical name>.orm"`. If not specified, this value is taken from the `kodo.Mapping` configuration property.
- `jdo-table`: This is an alias for the `kodo.jdo.jdbc.TableJDORMMappingFactory`. This factory stores mapping metadata as XML strings in a database table it creates for this purpose. It accepts the following options:
 - `ConstraintNames`: Set this option to `true` to include the names of foreign key and unique constraints in all generated mappings. By default, Kodo does not record constraint names.
 - `Table`: The name of the table. Defaults to `KODO_JDO_MAPPINGS`.
 - `NameColumn`: The name of the column that holds the mapping name. For class mappings, the mapping name is the class name. For named queries and sequences, it is the sequence or query name. Defaults to `NAME`.
 - `TypeColumn`: The name of the column that holds the mapping type. Defaults to `MAPPING_TYPE`. Type constants are:

- 0: A class mapping.
- 1: A named sequence.
- 2: A system-level named query.
- 3: A class-level named query.
- `MappingColumn`: The name of the column that holds the XML mapping. Defaults to `MAPPING_DEF`.

The mapping table is automatically created as needed, but you can also manipulate it through the `TableORMMappingFactory`'s `main` method, or the corresponding `mappingtable` shell/bat script. See the factory's **Javadoc** for usage instructions.

- `file`: Backwards-compatibility setting for Kodo 3.x mapping files. You can only use this factory with the `kodo3` metadata factory.
- `metadata`: Backwards-compatibility setting for Kodo 3.x mapping metadata extensions. You can only use this factory with the `kodo3` metadata factory.
- `db`: Backwards-compatibility setting for the Kodo 3.x mapping table. You can only use this factory with the `kodo3` metadata factory.

Example 7.19. Standard JPA Configuration

In the standard JPA configuration, the mapping factory is left unset.

```
<property name="kodo.MetadataFactory" value="jpa"/>
```

Example 7.20. Standard JDO Configuration

In the standard JDO configuration, the mapping factory is left unset.

```
kodo.MetadataFactory: jdo
```

Example 7.21. Recording Constraint Names

This example uses standard JDO mapping, but uses the `MappingFactory` property to tell Kodo to record the names of all generated constraints in the mappings.

```
kodo.jdbc.MappingFactory: ConstraintNames=true
```

Example 7.22. JDO ORM File Configuration

You can configure Kodo to separate JDO mapping data into `.orm` files either by setting the `kodo.Mapping` property to the logical mapping name, or by explicitly setting your `kodo.jdbc.MappingFactory` property. `unset`.

```
kodo.MetadataFactory: jdo
kodo.Mapping: oracle
```

```
kodo.MetadataFactory: jdo
kodo.jdbc.MappingFactory: jdo-orm(Mapping=oracle)
```

Example 7.23. Storing JDO Mappings in a Table

This example stores JDO mapping metadata in the `MAPPING` database table.

```
kodo.jdbc.MappingFactory: jdo-table(Table=MAPPING)
```

Example 7.24. JPA Metadata, JDO Mapping Files

This example configures Kodo to use JPA annotations for metadata, but JDO `.orm` files for mappings.

```
<property name="kodo.MetadataFactory" value="jpa"/>
<property name="kodo.jdbc.MappingFactory" value="jdo-orm"/>
```

7.5.1. Importing and Exporting Mapping Data

The **mapping tool** has the ability to import mapping data into the mapping factory, and to export mapping data from the mapping factory. Importing and exporting mapping data is useful for a couple of reasons. First, you may want to use a mapping factory that stores mapping data in an out-of-the-way location like the database, but you still want the ability to manipulate this information occasionally by hand. You can do so by exporting the data to XML, modifying it, and then re-importing it.

Example 7.25. Modifying Difficult-to-Access Mapping Data

```
mappingtool -a export -f mappings.xml package.jdo
... modify mappings.xml file as necessary ...
mappingtool -a import mappings.xml
```

Second, you can use the export/import facilities to switch mapping factories at any time.

Example 7.26. Switching Mapping Factories

```
mappingtool -a export -f mappings.xml *.jdo
... switch the kodo.jdbc.MappingFactory configuration ...
... property to list your new mapping factory choice ...
mappingtool -a import mappings.xml
```

Kodo uses JDO's `orm` mapping format for imports and exports. See [Chapter 15, Mapping Metadata \[260\]](#) for details on the `orm` mapping format.

7.6. Non-Standard Joins

The JPA Overview's [Chapter 12, Mapping Metadata \[118\]](#) and the JDO Overview's [Section 15.7, “Joins” \[269\]](#) explain join mapping in each specification. All of the examples in those documents, however, use “standard” joins, in that there is one foreign key column for each primary key column in the target table. Kodo supports additional join patterns, including partial primary key joins, non-primary key joins, and joins using constant values.

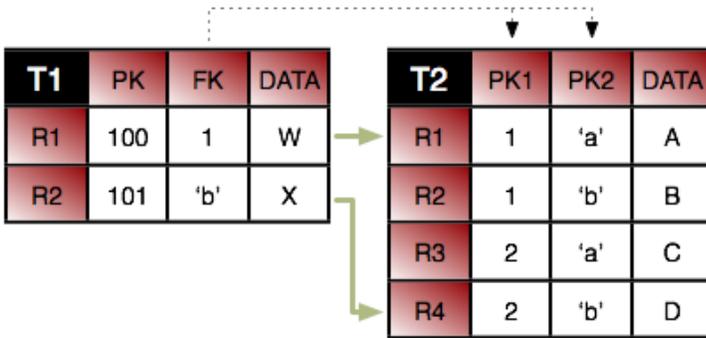
In a partial primary key join, the source table only has foreign key columns for a subset of the primary key columns in the target table. So long as this subset of columns correctly identifies the proper row(s) in the referenced table, Kodo will function properly. There is no special syntax for expressing a partial primary key join - just do not include column definitions for missing foreign key columns.

In a non-primary key join, at least one of the target columns is not a primary key. Once again, Kodo supports this join type with the same syntax as a primary key join. There is one restriction, however: each non-primary key column you are joining to must be controlled by a field mapping that implements the `kodo.jdbc.meta.Joinable` interface. All built in basic mappings implement this interface, including basic fields of embedded objects. Kodo will also respect any custom mappings that implement this interface. See [Section 7.10, “Custom Mappings” \[522\]](#) for an examination of custom mappings.

Not all joins consist of only links between columns. In some cases you might have a schema in which one of the join criteria is that a column in the source or target table must have some constant value. Kodo calls joins involving constant values *constant joins*.

To form a constant join in JPA mapping, first set the `JoinColumn`'s `name` attribute to the name of the column. If the column with the constant value is the target of the join, give its fully qualified name in the form `<table name>.<column name>`. Next, set the `referencedColumnName` attribute to the constant value. If the constant value is a string, place it in single quotes to differentiate it from a column name.

To form a constant join in JDO mapping, first set the `column` element's `name` attribute to the name of the column. If the column with the constant value is the target of the join, give its fully qualified name in the form `<table name>.<column name>`. Next, set the `target` attribute to the constant value. If the constant value is a string, place it in single quotes to differentiate it from a column name.



Consider the tables above. First, we want to join row T1 .R1 to row T2 .R1. If we just join column T1 .FK to T2 .PK1, we will wind up matching both T2 .R1 and T2 .R2. So in addition to joining T1 .FK to T2 .PK1, we also have to specify that T2 .PK2 has the value a. Here is how we'd accomplish this in mapping metadata.

JPA:

```
@Entity
@Table(name="T1")
public class ...
{
    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="FK" referencedColumnName="PK1"),
        @JoinColumn(name="T2.PK2" referencedColumnName="'a'")
    });
    private ...;
}
```

JDO:

```
<class name="..." table="T1">
  <...>
  <column name="FK" target="PK1"/>
  <column name="T2.PK2" target="'a'"/>
  </...>
</class>
```

Notice that we had to fully qualify the name of column PK2 because it is in the target table. Also notice that we put single quotes around the constant value so that it won't be confused with a column name. You do not need single quotes for numeric constants. For example, the syntax to join T1 .R2 to T2 .R4 is:

JPA:

```
@Entity
@Table(name="T1")
public class ...
{
    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="FK" referencedColumnName="PK2"),
        @JoinColumn(name="T2.PK1" referencedColumnName="2")
    });
    private ...;
}
```

JDO:

```
<class name="..." table="T1">
  <...>
    <column name="FK" target="PK2"/>
    <column name="T2.PK1" target="2"/>
  </...>
</class>
```

Finally, from the inverse direction, these joins would look like this:

JPA:

```
@Entity
@Table(name="T2")
public class ...
{
    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="T1.FK" referencedColumnName="PK1"),
        @JoinColumn(name="PK2" referencedColumnName="a' ")
    });
    private ...;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="T1.FK" referencedColumnName="PK2"),
        @JoinColumn(name="PK1" referencedColumnName="2")
    });
    private ...;
}
```

JDO:

```
<class name="..." table="T2">
  <...>
    <column name="T1.FK" target="PK1"/>
    <column name="PK2" target="a'"/>
  </...>
  <...>
    <column name="T1.FK" target="PK2"/>
    <column name="PK1" target="2"/>
  </...>
</class>
```

7.7. Additional JPA Mappings

Kodo supports many persistence strategies beyond those of the JPA specification. **Section 6.3, “Additional JPA Metadata” [481]** covered the logical metadata for Kodo's additional persistence strategies. We now demonstrate how to map entities using these strategies to the database.

7.7.1. Datastore Identity Mapping

Section 5.3, “Object Identity” [455] describes how to use datastore identity in JPA. Kodo requires a single numeric primary key column to hold datastore identity values. The `kodo.persistence.jdbc.DataStoreIdColumn` annotation customizes the datastore identity column. This annotation has the following properties:

- String name: Defaults to ID.

- int precision
- String columnDefinition
- boolean insertable
- boolean updatable

All properties correspond exactly to the same-named properties on the standard `Column` annotation, described in [Section 12.3, “Column” \[123\]](#).

Example 7.27. Datastore Identity Mapping

```
import kodo.persistence.*;
import kodo.persistence.jdbc.*;

@Entity
@Table(name="LOGS")
@DataStoreIdColumn(name="ENTRY")
public class LogEntry
{
    @Lob
    private String content;

    ...
}
```

7.7.2. Surrogate Version Mapping

Kodo supports version fields as defined by the JPA specification, but allows you to use a surrogate version column in place of a version field if you like. You map the surrogate version column with the `kodo.persistence.jdbc.VersionColumn` annotation. If you take advantage of Kodo's ability to define multiple **lock groups**, you may have multiple version columns. In that case, use the `kodo.persistence.jdbc.VersionColumns` annotation to declare an array of `VersionColumn` values. Each `VersionColumn` has the following properties:

- String name: Defaults to `VERSN`.
- String lockGroup
- int length
- int precision
- int scale
- String columnDefinition
- boolean nullable
- boolean insertable
- boolean updatable

The `lockGroup` property allows you to specify that a version column is for some lock group other than the default group. See

Section 5.8, “Lock Groups” [476] for an example. All other properties correspond exactly to the same-named properties on the standard `Column` annotation, described in **Section 12.3, “Column” [123]**.

By default, Kodo assumes that surrogate versioning uses a version number strategy. You can choose a different strategy with the `VersionStrategy` annotation described in **Section 7.9.1.4, “Version Strategy” [520]**

7.7.3. Multi-Column Mappings

Kodo makes it easy to create multi-column **custom mappings**. The JPA specification includes a `Column` annotation, but is missing a way to declare multiple columns for a single field. Kodo remedies this with the `kodo.persistence.jdbc.Columns` annotation, which contains an array of `Column` values. **Example 7.34, “Custom Mappings via Extensions” [524]** uses Kodo's `Columns` annotation to map a `java.awt.Point` to separate X and Y columns.

Remember to annotate custom field types with `Persistent`, as described in **Section 6.3.3, “Persistent Field Values” [482]**.

7.7.4. Join Column Attribute Targets

Section 12.8.4, “Direct Relations” [153] in the JPA Overview introduced you to the `JoinColumn` annotation. A `JoinColumn`'s `referencedColumnName` property declares which column in the table of the related type this join column links to. Suppose, however, that the related type is unmapped, or that it is part of a table-per-class inheritance hierarchy. Each subclass that might be assigned to the field could reside in a different table, and could use entirely different names for its primary key columns. It becomes impossible to supply a single `referencedColumnName` that works for all subclasses.

Kodo rectifies this by allowing you to declare which *attribute* in the related type each join column links to, rather than which column. If the attribute is mapped differently in various subclass tables, Kodo automatically forms the proper join for the subclass record at hand. The `kodo.persistence.jdbc.XJoinColumn` annotation has all the same properties as the standard `JoinColumn` annotation, but adds an additional `referencedAttributeName` property for this purpose. Simply use a `XJoinColumn` in place of a `JoinColumn` whenever you need to access this added functionality.

For compound keys, use the `kodo.persistence.jdbc.XJoinColumns` annotation. The value of this annotation is an array of individual `XJoinColumns`.

7.7.5. Embedded Mapping

JPA uses the `AttributeOverride` annotation to override the default mappings of an embeddable class. The JPA Overview details this process in **Section 12.8.3, “Embedded Mapping” [150]**. `AttributeOverrides` suffice for simple mappings, but do not allow you to override complex mappings. Also, JPA has no way to differentiate between a null embedded object and one with default values for all of its fields.

Kodo overcomes these shortcomings with the `kodo.persistence.jdbc.EmbeddedMapping` annotation. This annotation has the following properties:

- `String nullIndicatorColumnName`: If the named column's value is `NULL`, then the embedded object is assumed to be null. If the named column has a non-`NULL` value, then the embedded object will get loaded and populated with data from the other embedded fields. This property is entirely optional. By default, Kodo always assumes the embedded object is non-null, just as in standard JPA mapping.

If the column you name does not belong to any fields of the embedded object, Kodo will create a synthetic null-indicator column with this name. In fact, you can specify a value of `true` to simply indicate that you want a synthetic null-indicator column, without having to come up with a name for it. A value of `false` signals that you explicitly do not want a null-indicator column created for this mapping (in case you have configured your **mapping defaults** to create one by default).

- `String nullIndicatorFieldName`: Rather than name a null indicator column, you can name a field of the embedded type. Kodo will use the column of this field as the null-indicator column.
- `MappingOverride[] overrides`: This array allows you to override any mapping of the embedded object.

The `EmbeddedMapping`'s `overrides` array serves the same purpose as standard JPA's `AttributeOverrides` and `AssociationOverrides`. In fact, you can also use the `MappingOverride` annotation on an entity class to override a complex mapping of its mapped superclass, just as you can with `AttributeOverride` and `AssociationOverrides`. The `MappingOverrides` annotation, whose value is an array of `MappingOverrides`, allows you to override multiple mapped superclass mappings.

Each `kodo.persistence.jdbc.MappingOverride` annotation has the following properties:

- `String name`: The name of the field that is being overridden.
- `Column[] columns`: Columns for the new field mapping.
- `XJoinColumn[] joinColumns`: Join columns for the new field mapping, if it is a relation field.
- `ContainerTable containerTable`: Table for the new collection or map field mapping. We cover collection mappings in [Section 7.7.6, “Collections” \[512\]](#) and map mappings in [Section 7.7.8, “Maps” \[515\]](#)
- `ElementColumn[] elementColumns`: Element columns for the new collection or map field mapping. You will see how to use element columns in [Section 7.7.6.2, “Element Columns” \[512\]](#)
- `ElementJoinColumn[] elementJoinColumns`: Element join columns for the new collection or map field mapping. You will see how to use element join columns in [Section 7.7.6.3, “Element Join Columns” \[512\]](#)
- `KeyColumn[] keyColumns`: Map key columns for the new map field mapping. You will see how to use key columns in [Section 7.7.8.1, “Key Columns” \[515\]](#)
- `KeyJoinColumn[] keyJoinColumns`: Key join columns for the new map field mapping. You will see how to use key join columns in [Section 7.7.8.2, “Key Join Columns” \[516\]](#)

The following example defines an embeddable `PathCoordinate` class with a custom mapping of a `java.awt.Point` field to two columns. It then defines an entity which embeds a `PathCoordinate` and overrides the default mapping for the `point` field. The entity also declares that if the `PathCoordinate`'s `siteName` field column is null, it means that no `PathCoordinate` is stored in the embedded record; the owning field will load as null.

Example 7.28. Overriding Complex Mappings

```
import kodo.persistence.jdbc.*;

@Embeddable
public class PathCoordinate
{
    private String siteName;

    @Persistent
    @Strategy("com.xyz.kodo.PointValueHandler")
    private Point point;

    ...
}

@Entity
public class Path
{
    @Embedded
    @EmbeddedMapping(nullIndicatorFieldName="siteName", overrides={
        @MappingOverride(name="siteName", columns=@Column(name="START_SITE")),
        @MappingOverride(name="point", columns={
            @Column(name="START_X"),
            @Column(name="START_Y")
        })
    })
    private PathCoordinate start;

    ...
}
```

7.7.6. Collections

In [Section 6.3.4, “Persistent Collection Fields” \[482\]](#), we explored the `PersistentCollection` annotation for persistent collection fields that aren't a standard `OneToMany` or `ManyToMany` relation. To map these non-standard collections, combine Kodo's `ContainerTable` annotation with `ElementColumns`, `ElementJoinColumns`, or an `ElementEmbeddedMapping`. We explore the annotations below.

7.7.6.1. Container Table

The `kodo.persistence.jdbc.ContainerTable` annotation describes a database table that holds collection (or map) elements. This annotation has the following properties:

- `String` name
- `String` catalog
- `String` schema
- `XJoinColumn[]` joinColumns
- `ForeignKey` joinForeignKey
- `Index` joinIndex

The `name`, `catalog`, `schema`, and `joinColumns` properties describe the container table and how it joins to the owning entity's table. These properties correspond to the same-named properties on the standard `JoinTable` annotation, described in [Section 12.8.5, “Join Table” \[156\]](#). If left unspecified, the name of the table defaults to the first five characters of the entity table name, plus an underscore, plus the field name. The `joinForeignKey` and `joinIndex` properties override default foreign key and index generation for the join columns. We explore foreign keys and indexes later in this chapter.

You may notice that the container table does not define how to store the collection elements. That is left to separate annotations, which are the subject of the next sections.

7.7.6.2. Element Columns

Just as the JPA `Column` annotation maps a simple value (primitive wrapper, `String`, etc), Kodo's `kodo.persistence.jdbc.ElementColumn` annotation maps a simple element value. To map custom multi-column elements, use the `kodo.persistence.jdbc.ElementColumns` annotation, whose value is an array of `ElementColumns`.

An `ElementColumn` always resides in a container table, so it does not have the `table` property of a standard `Column`. Otherwise, the `ElementColumn` and standard `Column` annotations are equivalent. See [Section 12.3, “Column” \[123\]](#) in the JPA Overview for a review of the `Column` annotation.

7.7.6.3. Element Join Columns

Element join columns are equivalent to standard JPA join columns, except that they represent a join to a collection or map element entity rather than a direct relation. You represent an element join column with Kodo's `kodo.persistence.jdbc.ElementJoinColumn` annotation. To declare a compound join, enclose an array of `ElementJoinColumns` in the `kodo.persistence.jdbc.ElementJoinColumns` annotation.

An `ElementJoinColumn` always resides in a container table, so it does not have the `table` property of a standard `JoinColumn`. Like `XJoinColumns` above, `ElementJoinColumns` can reference a linked attribute rather than a static linked column. Otherwise, the `ElementJoinColumn` and standard `JoinColumn` annotations are equivalent. See [Section 12.8.4, “Direct Relations” \[153\]](#) in the JPA Overview for a review of the `JoinColumn` annotation.

7.7.6.4. Element Embedded Mapping

The `kodo.persistence.jdbc.ElementEmbeddedMapping` annotation allows you to map your collection or map's embedded element type to your container table. This annotation has exactly the same properties as the `EmbeddedMapping` annotation described [above](#).

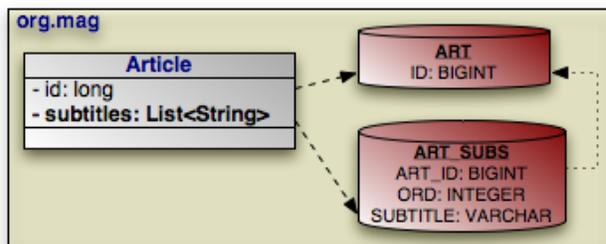
7.7.6.5. Order Column

Relational databases do not guarantee that records are returned in insertion order. If you want to make sure that your collection elements are loaded in the same order they were in when last stored, you must declare an order column. Kodo's `kodo.persistence.jdbc.OrderColumn` annotation has the following properties:

- String name: Defaults to ORDR.
- boolean enabled
- int precision
- String columnDefinition
- boolean insertable
- boolean updatable

Order columns are always in the container table. You can explicitly turn off ordering (if you have enabled it by default via your **mapping defaults**) by setting the `enabled` property to `false`. All other properties correspond exactly to the same-named properties on the standard `Column` annotation, described in [Section 12.3, "Column" \[123\]](#).

7.7.6.6. Examples



Our first example maps the `Article.subtitles` field to the `ART_SUBS` container table, as shown in the diagram above. Notice the use of `ContainerTable` in combination with `ElementColumn` and `OrderColumn` to map this ordered list of strings.

Example 7.29. String List Mapping

```
package org.mag;

import kodo.persistence.*;
import kodo.persistence.jdbc.*;

@Entity
@Table(name="ART")
public class Article
{
    @Id private long id;

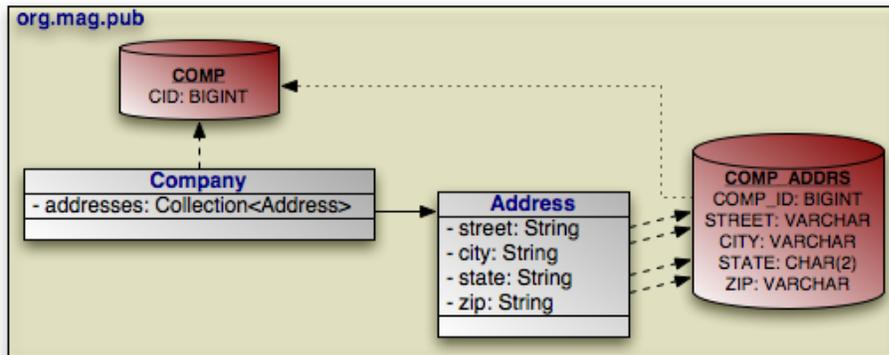
    @PersistentCollection
    @ContainerTable(name="ART_SUBS", joinColumns=@JoinColumn(name="ART_ID"))
    @ElementColumn(name="SUBTITLE")
    @OrderColumn(name="ORD")
}
```

```

private List<String> subtitles;
    ...
}

```

Now we map a collection of embedded `Address` objects for a `Company`, according to the following diagram:



Example 7.30. Embedded Element Mapping

```

package org.mag.pub;

import kodo.persistence.*;
import kodo.persistence.jdbc.*;

@Embeddable
public class Address
{
    ...
}

@Entity
@Table(name="COMP")
public class Company
{
    @Id private long id;

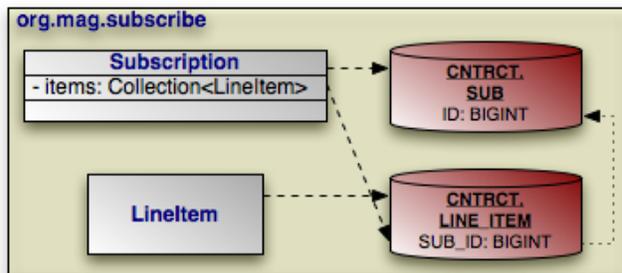
    @PersistentCollection(elementEmbedded=true)
    @ContainerTable(name="COMP_ADDR", joinColumns=@XJoinColumn(name="COMP_ID"))
    @ElementEmbeddedMapping(overrides=@MappingOverride(name="state",
        columns=@Column(columnDefinition="CHAR(2)")))
    private Collection<Address> addresses;

    ...
}

```

7.7.7. One-Sided One-Many Mapping

The previous section covered the use of `ElementJoinColumn` annotations in conjunction with a `ContainerTable` for mapping collections to dedicated tables. `ElementJoinColumns`, however, have one additional use: to create a one-sided one-many mapping. Standard JPA supports `OneToMany` fields without a `mappedBy` inverse, but only by mapping these fields to a `JoinTable` (see [Section 12.8.5, “Join Table” \[156\]](#) in the JPA Overview for details). Often, you'd like to create a one-many association based on an inverse foreign key (logical or actual) in the table of the related type.



Consider the model above. `Subscription` has a collection of `LineItem`s, but `LineItem` has no inverse relation to `Subscription`. To retrieve all of the `LineItem` records for a `Subscription`, we join the `SUB_ID` inverse foreign key column in the `LINE_ITEM` table to the primary key column of the `SUB` table. The example below shows how to represent this model in mapping annotations. Note that Kodo automatically assumes an inverse foreign key mapping when element join columns are given, but no container or join table is given.

Example 7.31. One-Sided One-Many Mapping

```
package org.mag.subscribe;

import kodo.persistence.jdbc.*;

@Entity
@Table(name="LINE_ITEM", schema="CNTRCT")
public class LineItem
{
    ...
}

@Entity
@Table(name="SUB", schema="CNTRCT")
public class Subscription
{
    @Id private long id;

    @OneToMany
    @ElementJoinColumn(name="SUB_ID", target="ID")
    private Collection<LineItem> items;

    ...
}
```

7.7.8. Maps

Section 6.3.5, “Persistent Map Fields” [483] discussed the `PersistentMap` annotation for persistent map fields. To map these non-standard fields to the database, combine Kodo's `ContainerTable` annotation with `KeyColumns`, `KeyJoinColumn`s, or an `KeyEmbeddedMapping` and `ElementColumns`, `ElementJoinColumn`s, or an `ElementEmbeddedMapping`.

We detailed the `ContainerTable` annotation in **Section 7.7.6.1, “Container Table” [512]**. We also discussed element columns, join columns, and embedded mappings in **Section 7.7.6.2, “Element Columns” [512]**, **Section 7.7.6.3, “Element Join Columns” [512]**, and **Section 7.7.6.4, “Element Embedded Mapping” [512]**. Key columns, join columns, and embedded mappings are new, however; we tackle them below.

7.7.8.1. Key Columns

Key columns serve the same role for map keys as the element columns described in **Section 7.7.6.2, “Element Columns” [512]** serve for collection elements. Kodo's `kodo.persistence.jdbc.KeyColumn` annotation represents a map key. To map

custom multi-column keys, use the `kodo.persistence.jdbc.KeyColumns` annotation, whose value is an array of `KeyColumns`.

A `KeyColumn` always resides in a container table, so it does not have the `table` property of a standard `Column`. Otherwise, the `KeyColumn` and standard `Column` annotations are equivalent. See [Section 12.3, “Column” \[123\]](#) in the JPA Overview for a review of the `Column` annotation.

7.7.8.2. Key Join Columns

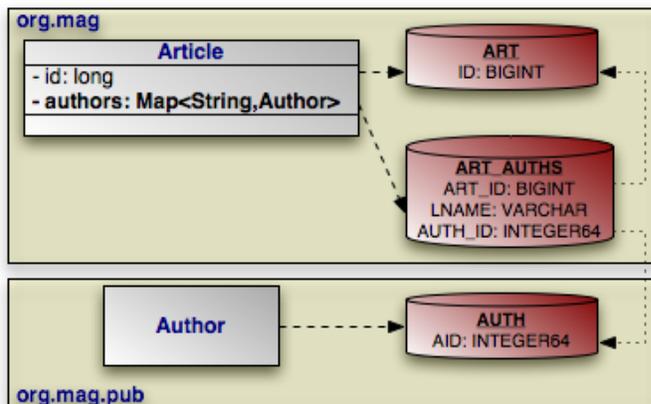
Key join columns are equivalent to standard JPA join columns, except that they represent a join to a map key entity rather than a direct relation. You represent a key join column with Kodo's `kodo.persistence.jdbc.KeyJoinColumn` annotation. To declare a compound join, enclose an array of `KeyJoinColumns` in the `kodo.persistence.jdbc.KeyJoinColumns` annotation.

A `KeyJoinColumn` always resides in a container table, so it does not have the `table` property of a standard `JoinColumn`. Like `XJoinColumns` above, `KeyJoinColumns` can reference a linked field rather than a static linked column. Otherwise, the `KeyJoinColumn` and standard `JoinColumn` annotations are equivalent. See [Section 12.8.4, “Direct Relations” \[153\]](#) in the JPA Overview for a review of the `JoinColumn` annotation.

7.7.8.3. Key Embedded Mapping

The `kodo.persistence.jdbc.KeyEmbeddedMapping` annotation allows you to map your map field's embedded key type to your container table. This annotation has exactly the same properties as the `EmbeddedMapping` annotation described above.

7.7.8.4. Examples



Map mapping in Kodo uses the same principles you saw in collection mapping. The example below maps the `Article.authors` map according to the diagram above.

Example 7.32. String Key, Entity Value Map Mapping

```
package org.mag.pub;

import kodo.persistence.*;
import kodo.persistence.jdbc.*;

@Entity
@Table(name="AUTH")
@DataStoreIdColumn(name="AID" columnDefinition="INTEGER64")
public class Author
{
    ...
}
```

```

package org.mag;

@Entity
@Table(name="ART")
public class Article
{
    @Id private long id;

    @PersistentMap
    @ContainerTable(name="ART_AUTHS", joinColumns=@XJoinColumn(name="ART_ID"))
    @KeyColumn(name="LNAME")
    @ElementJoinColumn(name="AUTH_ID")
    private Map<String,Author> authors;

    ...
}

```

7.7.9. Indexes and Constraints

Kodo uses index information during schema generation to index the proper columns. Kodo uses foreign key and unique constraint information during schema creation to generate the proper database constraints, and also at runtime to order SQL statements to avoid constraint violations while maximizing SQL batch size.

Kodo assumes certain columns have indexes or constraints based on your mapping defaults, as detailed in [Section 7.4, “Mapping Defaults” \[500\]](#). You can override the configured defaults on individual joins, field values, collection elements, map keys, or map values using the annotations presented in the following sections.

7.7.9.1. Indexes

The `kodo.persistence.jdbc.Index` annotation represents an index on the columns of a field. It is also used within the `ContainerTable` annotation to index join columns.

To index the columns of a collection or map element or map key, use the `kodo.persistence.jdbc.ElementIndex` and `kodo.persistence.jdbc.KeyIndex` annotations, respectively. These annotations have the following properties:

- `boolean enabled`: Set this property to `false` to explicitly tell Kodo not to index these columns, when Kodo would otherwise do so.
- `String name`: The name of the index. Kodo will choose a name if you do not provide one.
- `boolean unique`: Whether to create a unique index. Defaults to `false`.

7.7.9.2. Foreign Keys

The `kodo.persistence.jdbc.ForeignKey` annotation represents a foreign key on the columns of a field. It is also used within the `ContainerTable` annotation to set a database foreign key on join columns.

To set a constraint to the columns of a collection or map element or map value, use the `kodo.persistence.jdbc.ElementForeignKey` and `kodo.persistence.jdbc.KeyForeignKey` annotations, respectively. These annotations have the following properties:

- `boolean enabled`: Set this property to `false` to explicitly tell Kodo not to set a foreign key on these columns, when Kodo would otherwise do so.
- `String name`: The name of the foreign key. Kodo will choose a name if you do not provide one, or will create an anonymous key.

- `boolean deferred`: Whether to create a deferred key if supported by the database.
- `ForeignKeyAction deleteAction`: Value from the `kodo.persistence.jdbc.ForeignKeyAction` enum identifying the desired delete action. Defaults to `RESTRICT`.
- `ForeignKeyAction updateAction`: Value from the `kodo.persistence.jdbc.ForeignKeyAction` enum identifying the desired update action. Defaults to `RESTRICT`.

Keep in mind that Kodo uses foreign key information at runtime to avoid constraint violations; it is important, therefore, that your **mapping defaults** and foreign key annotations combine to accurately reflect your existing database constraints.

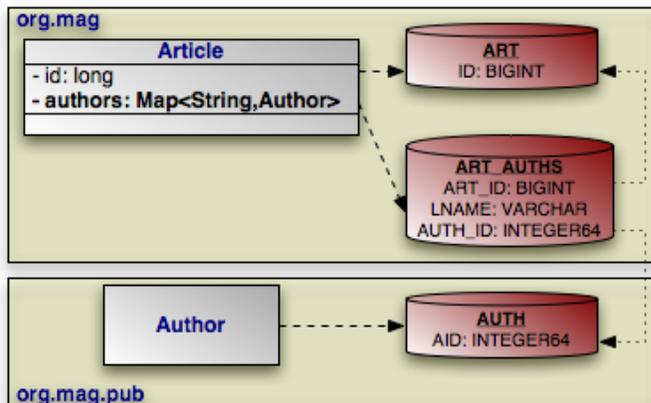
7.7.9.3. Unique Constraints

The `kodo.persistence.jdbc.Unique` annotation represents a unique constraint on the columns of a field. It is more convenient than using the `uniqueConstraints` property of standard JPA `Table` and `SecondaryTable` annotations, because you can apply it directly to the constrained field. The `Unique` annotation has the following properties:

- `boolean enabled`: Set this property to `false` to explicitly tell Kodo not to constrain these columns, when Kodo would otherwise do so.
- `String name`: The name of the constraint. Kodo will choose a name if you do not provide one, or will create an anonymous constraint.
- `boolean deferred`: Whether to create a deferred constraint if supported by the database.

7.7.9.4. Examples

Here again is our map example from [Section 7.7.8, “Maps” \[515\]](#), now with explicit indexes and constraints added.



Example 7.33. Constraint Mapping

```
package org.mag.pub;

import kodo.persistence.*;
import kodo.persistence.jdbc.*;

@Entity
@Table(name="AUTH")
@DataStoreIdColumn(name="AID" columnDefinition="INTEGER64")
public class Author
```

```

{
  ...
}

package org.mag;

@Entity
@Table(name="ART")
public class Article
{
    @Id private long id;

    @PersistentMap
    @ContainerTable(name="ART_AUTHS", joinColumns=@XJoinColumn(name="ART_ID")
        joinForeignKey=@ForeignKey(deleteAction=ForeignKeyAction.CASCADE))
    @KeyColumn(name="LNAME")
    @KeyIndex(name="I_AUTH_LNAME")
    @ElementJoinColumn(name="AUTH_ID")
    @ElementForeignKey(deleteAction=ForeignKeyAction.RESTRICT)
    private Map<String,Author> authors;

    ...
}

```

7.8. Mapping Limitations

The following sections outline the limitations Kodo places on specific mapping strategies.

7.8.1. Table Per Class

Table-per-class inheritance mapping has the following limitations:

- You cannot traverse polymorphic relations to non-leaf classes in a table-per-class inheritance hierarchy in queries.
- You cannot map a one-sided polymorphic relation to a non-leaf class in a table-per-class inheritance hierarchy using an inverse foreign key.
- You cannot use an order column in a polymorphic relation to a non-leaf class in a table-per-class inheritance hierarchy mapped with an inverse foreign key.
- Table-per-class hierarchies impose limitations on eager fetching. See [Section 5.7.2, “Eager Fetching Considerations and Limitations” \[475\]](#).

Note

Non-polymorphic relations do not suffer from these limitations. You can declare a non-polymorphic relation using the extensions described in [Section 7.9.2.2, “Nonpolymorphic” \[520\]](#)

7.9. Mapping Extensions

Mapping extensions allow you to access Kodo-specific functionality from your mappings. Note that all extensions below are specific to mappings. If you store your mappings separately from your persistence metadata, these extensions must be specified along with the mapping information, not the persistence metadata information.

It is only valid to place these extensions on JDO persistence metadata if you store your mappings in your `.jdo` files, or if you want these extensions to act as mapping hints (see [Section 7.1.3, “JDO Forward Mapping Hints” \[494\]](#)). All extension keys use a `vendor-name` of `kodo`.

7.9.1. Class Extensions

Kodo recognizes the following class extensions.

7.9.1.1. Subclass Fetch Mode

This extension specifies how to eagerly fetch subclass state. It overrides the global `kodo.jdbc.SubclassFetchMode` property. Set the JPA `kodo.persistence.jdbc.SubclassFetchMode` annotation to a value from the `kodo.persistence.jdbc.EagerFetchType` enum: `JOIN`, `PARALLEL`, or `NONE`. Set the JDO `jdbc-subclass-fetch-mode` XML extension key to `join`, `parallel`, or `none`. See [Section 5.7, “Eager Fetching” \[473\]](#) for a discussion of eager fetching.

7.9.1.2. Strategy

The `kodo.persistence.jdbc.Strategy` class annotation allows you to specify a custom mapping strategy for your class. See [Section 7.10, “Custom Mappings” \[522\]](#) for information on custom mappings. Note that this extension applies only to JPA mapping; when using JDO mappings you specify custom class strategies using the standard XML inheritance element's `strategy` attribute.

7.9.1.3. Discriminator Strategy

The `kodo.persistence.jdbc.DiscriminatorStrategy` class annotation allows you to specify a custom discriminator strategy. See [Section 7.10, “Custom Mappings” \[522\]](#) for information on custom mappings. Note that this extension applies only to JPA mapping; when using JDO mappings you specify custom discriminator strategies using the standard XML discriminator element's `strategy` attribute.

7.9.1.4. Version Strategy

The `kodo.persistence.jdbc.VersionStrategy` class annotation allows you to specify a custom version strategy. See [Section 7.10, “Custom Mappings” \[522\]](#) for information on custom mappings. Note that this extension applies only to JPA mapping; when using JDO mappings you specify custom version strategies using the standard XML `version` element's `strategy` attribute.

7.9.2. Field Extensions

Kodo recognizes the following field extensions.

7.9.2.1. Eager Fetch Mode

This extension specifies how to eagerly fetch related objects. It overrides the global `kodo.jdbc.EagerFetchMode` property. Set the JPA `kodo.persistence.jdbc.EagerFetchMode` annotation to a value from the `kodo.persistence.jdbc.EagerFetchType` enum: `JOIN`, `PARALLEL`, or `NONE`. Set the JDO `jdbc-eager-fetch-mode` XML extension key to `join`, `parallel`, or `none`. See [Section 5.7, “Eager Fetching” \[473\]](#) for a discussion of eager fetching.

7.9.2.2. Nonpolymorphic

All fields in Java are polymorphic. If you declare a field of type `T`, you can assign any subclass of `T` to the field as well. This is very convenient, but can make relation traversal very inefficient under some inheritance strategies. It can even make querying across the field impossible. Often, you know that certain fields do not need to be entirely polymorphic. By telling Kodo about such fields, you can improve the efficiency of your relations.

Note

Kodo also includes the `type` metadata extension for narrowing the declared type of a field See [Section 6.4.2.7, “Type” \[486\]](#).

Kodo JPA defines the following extensions for nonpolymorphic values:

- `kodo.persistence.jdbc.Nonpolymorphic`
- `kodo.persistence.jdbc.ElementNonpolymorphic`
- `kodo.persistence.jdbc.KeyNonpolymorphic`

The value of these extensions is a constant from the `kodo.persistence.jdbc.NonpolymorphicType` enumeration. The default value, `EXACT`, indicates that the relation will always be of the exact declared type. A value of `JOINABLE`, on the other hand, means that the relation might be to any joinable subclass of the declared type. This value only excludes table-per-class subclasses.

Kodo JDO defines the following nonpolymorphic mapping extension keys:

- `jdbc-nonpolymorphic`
- `jdbc-element-nonpolymorphic`
- `jdbc-key-nonpolymorphic`
- `jdbc-value-nonpolymorphic`

These keys apply to field values, collection or array elements, map keys, and map values, respectively. Set the extension value to `exact` to indicate that the relation is always to the exact declared type. Set it to `joinable` if the relation can be to any joinable subclass of the declared type. A value of `joinable` only excludes table-per-class subclasses.

7.9.2.3. Class Criteria

This family of boolean extensions determines whether Kodo will use the expected class of related objects as criteria in the SQL it issues to load a relation field. Typically, this is not needed. The foreign key values uniquely identify the record for the related object. Under some rare mappings, however, you may need to consider both foreign key values and the expected class of the related object - for example, if you have an inverse relation that shares the foreign key with another inverse relation to an object of a different subclass. In these cases, set the proper class criteria extension to `true` to force Kodo to append class criteria to its select SQL.

Kodo JPA defines the following class criteria annotations for field relations, array, collection, and map element relations, and map key relations, respectively:

- `kodo.persistence.jdbc.ClassCriteria`
- `kodo.persistence.jdbc.ElementClassCriteria`
- `kodo.persistence.jdbc.KeyClassCriteria`

Kodo JDO defines the following class criteria mapping extension keys for field relations, collection and array element relations, map key relations, and map value relations, respectively:

- `jdbc-class-criteria`

- `jdbc-element-class-criteria`
- `jdbc-key-class-criteria`
- `jdbc-value-class-criteria`

7.9.2.4. Strategy

Kodo's family of strategy extensions allow you to specify a custom mapping strategy or value handler for a field. See [Section 7.10, “Custom Mappings” \[522\]](#) for information on custom mappings.

Kodo includes the following JPA strategy annotations:

- `kodo.persistence.jdbc.Strategy` : Field strategy or value handler plugin string.
- `kodo.persistence.jdbc.ElementStrategy` : Array, collection, or map element value handler plugin string.
- `kodo.persistence.jdbc.KeyStrategy` : Map key value handler plugin string.

Kodo recognizes the following JDO strategy extension keys:

- `jdbc-strategy`: Field strategy or value handler plugin string.
- `jdbc-element-strategy`: Collection or array element value handler plugin string.
- `jdbc-key-strategy`: Map key value handler plugin string.
- `jdbc-value-strategy`: Map value handler plugin string.

7.9.3. Column Extensions

Kodo recognizes the following extensions within JDO `column` elements:

7.9.3.1. insertable

Boolean extension dictating whether the current mapping can insert into this column.

7.9.3.2. updatable

Boolean extension dictating whether the current mapping can update this column value.

7.9.3.3. lock-group

This extension is only valid on version columns. When you use custom lock groups, it allows you to declare which version column is for which lock group. See [Section 5.8, “Lock Groups” \[476\]](#).

7.10. Custom Mappings

In Kodo, you are not limited to the set of standard mappings defined by the specification. Kodo allows you to define custom class, discriminator, version, and field mapping strategies with all the power of Kodo's built-in strategies. [Section 1.3.5, “Custom Mappings” \[62\]](#) describes custom mapping samples that ship with Kodo.

7.10.1. Custom Class Mapping

To create a custom class mapping, write an implementation of the `kodo.jdbc.meta.ClassStrategy` interface. You will probably want to extend one of the existing abstract or concrete strategies in the `kodo.jdbc.meta.strats` package.

Important

This feature requires **Kodo Enterprise Edition**.

The `kodo.persistence.jdbc.Strategy` annotation allows you to declare a custom class mapping strategy in JPA mapping metadata. Set the value of the annotation to the full class name of your custom strategy. You specify custom strategies in JDO by setting the `inheritance` element's `strategy` attribute to the full class name of your implementation. You can configure your strategy class' bean properties using Kodo's plugin syntax, detailed in [Section 2.5, “Plugin Configuration” \[395\]](#).

7.10.2. Custom Discriminator and Version Strategies

To define a custom discriminator or version strategy, implement the `kodo.jdbc.meta.DiscriminatorStrategy` or `kodo.jdbc.meta.VersionStrategy` interface, respectively. You might extend one of the existing abstract or concrete strategies in the `kodo.jdbc.meta.strats` package.

Kodo includes the `kodo.persistence.jdbc.DiscriminatorStrategy` and `kodo.persistence.jdbc.VersionStrategy` class annotations for declaring a custom discriminator or version strategy in JPA mapping metadata. Set the string value of these annotations to the full class name of your implementation, or to the class name or alias of an existing Kodo implementation.

To specify your custom strategy in JDO metadata, set the appropriate `strategy` attribute to the full class name of your implementation. To use a custom discriminator strategy, specify its class name in the `discriminator` element's `strategy` attribute. To use a custom version strategy, specify its class name in the `version` element's `strategy` attribute.

As with custom class mappings, you can configure your strategy class' bean properties using Kodo's plugin syntax, detailed in [Section 2.5, “Plugin Configuration” \[395\]](#).

7.10.3. Custom Field Mapping

While custom class, discriminator, and version mapping can be useful, custom field mappings are far more common. Kodo offers two types of custom field mappings: value handlers, and full custom field strategies. The following sections examine each.

7.10.3.1. Value Handlers

Value handlers make it trivial to map any type that you can break down into one or more simple values. All value handlers implement the `kodo.jdbc.meta.ValueHandler` interface; see its **Javadoc** for details. Rather than give synthetic examples of value handlers, Kodo provides the source to most of the built-in handlers in the `src/kodo/jdbc/meta/strats` directory of your Kodo distribution. Use these functional implementations as examples when you create your own value handlers.

Note that value handlers are not only simple to write, but are highly reusable. For example, imagine that you create a handler for `java.awt.Point` values. You can not only use this handler to map fields of type `Point`, but also to map `Collections` or `Maps` of `Points` with no additional work.

7.10.3.2. Field Strategies

Kodo interacts with persistent fields through the `kodo.jdbc.meta.FieldStrategy` interface. You can implement this interface yourself to create a custom field strategy, or extend one of the existing abstract or concrete strategies in the `kodo.jdbc.meta.strats` package. Creating a custom field strategy is more difficult than writing a custom value handler, but gives you more freedom in how you interact with the database. [Section 1.3.5, “Custom Mappings” \[62\]](#) describes custom mapping samples that ship with Kodo.

7.10.3.3. Configuration

Kodo gives you two ways to configure your custom field mappings. The `FieldStrategies` property of the built-in `MappingDefaults` implementations allows you to globally associate field types with their corresponding custom value handler or strategy. Kodo will automatically use your custom strategies when it encounters a field of the associated type. Kodo will use your custom value handlers whenever it encounters a field, collection element, map key, or map value of the associated type. [Section 7.4, “Mapping Defaults” \[500\]](#) described mapping defaults in detail.

Your other option is to explicitly install a custom value handler or strategy on a particular field. To do so, specify the full name of your implementation class in the proper mapping metadata extension. Kodo JPA includes the `kodo.persistence.jdbc.Strategy`, `kodo.persistence.jdbc.ElementStrategy`, and `kodo.persistence.jdbc.KeyStrategy` annotations. Kodo JDO recognizes `jdbc-strategy`, `jdbc-element-strategy`, `jdbc-key-strategy`, and `jdbc-value-strategy` field extension keys. You can configure the named strategy or handler's bean properties in these extensions using Kodo's plugin format (see [Section 2.5, “Plugin Configuration” \[395\]](#)).

The example below installs a custom strategy on the `coverImage` field, uses a custom value handler for the `primaryInfoStruct` field, and uses the same value handler for the elements of the `secondaryInfoStructs` collection.

Example 7.34. Custom Mappings via Extensions

JPA:

```
import kodo.persistence.*;
import kodo.persistence.jdbc.*;

@Entity
public class Magazine
{
    @Persistent
    @Strategy("org.mag.mapping.ImageStrategy")
    @Column(name="IMG")
    private Image coverImage;

    @Persistent
    @Strategy("org.mag.mapping.InfoStructHandler")
    @Columns({
        @Column(name="DATA1"),
        @Column(name="DATA2"),
        @Column(name="DATA3")
    })
    private InfoStruct primaryInfoStruct;

    @PersistentCollection
    @ContainerTable(name="SEC_STRUCTS", joinColumns=@XJoinColumn(name="MAG_ID"))
    @ElementStrategy("org.mag.mapping.InfoStructHandler")
    @ElementColumns({
        @ElementColumn(name="DATA1"),
        @ElementColumn(name="DATA2"),
        @ElementColumn(name="DATA3")
    })
    private Collection<InfoStruct> secondaryInfoStructs;

    ...
}
```

JDO:

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag">
    <class name="Magazine" table="MAG">
      <field name="coverImage" column="IMG">
        <extension vendor-name="kodo" key="jdbc-strategy"
          value="org.mag.mapping.ImageStrategy"/>
      </field>
      <field name="primaryInfoStruct">
        <column name="DATA1"/>
      </field>
    </class>
  </package>
</orm>
```

```

        <column name="DATA2" />
        <column name="DATA3" />
        <extension vendor-name="kodo" key="jdbc-strategy"
            value="org.mag.mapping.InfoStructHandler" />
    </field>
    <field name="secondaryInfoStructs" table="SEC_STRUCTS">
        <join column="MAG_ID" />
        <element>
            <column name="DATA1" />
            <column name="DATA2" />
            <column name="DATA3" />
        </element>
        <extension vendor-name="kodo" key="jdbc-element-strategy"
            value="org.mag.mapping.InfoStructHandler" />
    </field>
    ..
</class>
</package>
</orm>

```

7.11. Orphaned Keys

Unless you apply database foreign key constraints extensively, it is possible to end up with orphaned keys in your database. For example, suppose Magazine *m* has a reference to Article *a*. If you delete *a* without nulling *m*'s reference, *m*'s database record will wind up with an orphaned key to the non-existent *a* record.

Note

One way of avoiding orphaned keys is to use *dependent* fields.

Kodo's `kodo.OrphanedKeyAction` configuration property controls what action to take when Kodo encounters an orphaned key. You can set this plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) to a custom implementation of the `kodo.event.OrphanedKeyAction` interface, or use one of the built-in options:

- `log`: This is the default setting. This option logs a message for each orphaned key. It is an alias for the `kodo.event.LogOrphanedKeyAction` class, which has the following additional properties:
 - `Channel`: The channel to log to. Defaults to `kodo.Runtime`.
 - `Level`: The level to log at. Defaults to `WARN`.
- `exception`: Throw an exception when Kodo discovers an orphaned key. This is an alias for the `kodo.event.ExceptionOrphanedKeyAction` class.

In JPA, the exception type will be `javax.persistence.EntityNotFoundException`.

In JDO, the exception type will be `javax.jdo.JDOObjectNotFoundException`.

- `none`: Ignore orphaned keys. This is an alias for the `kodo.event.NoneOrphanedKeyAction` class.

Example 7.35. Custom Logging Orphaned Keys

JPA XML format:

```
<property name="kodo.OrphanedKeyAction" value="log(Channel=Orphans, Level=DEBUG)" />
```

JDO properties format:

```
kodo.OrphanedKeyAction: log(Channel=Orphans, Level=DEBUG)
```

Chapter 8. Deployment

Kodo deployment includes choosing a factory deployment strategy, and in a managed environment, optionally integrating with your application server's managed and XA transactions. This chapter examines each aspect of deployment in turn.

8.1. Factory Deployment

Kodo offers several `EntityManagerFactory` and `PersistenceManagerFactory` deployment options.

8.1.1. Standalone Deployment

The JPA Overview describes the `javax.persistence.Persistence` class. You can use `Persistence` to obtain `EntityManagerFactory` instances, as demonstrated in [Chapter 6, Persistence \[63\]](#). Kodo also extends `Persistence` to add additional `EntityManagerFactory` creation methods. The `kodo.persistence.KodoPersistence` class **Javadoc** details these extensions.

[Chapter 6, JDOHelper \[201\]](#) of the JDO Overview describes the `JDOHelper` class. [Section 6.3, “PersistenceManagerFactory Construction” \[205\]](#) shows how to use the `JDOHelper` to construct `PersistenceManagerFactory` objects in a vendor-neutral fashion.

After obtaining the factory, you can cache it for all `EntityManager` or `PersistenceManager` creation duties.

8.1.2. EntityManager Injection

To be decided.

8.1.3. Kodo JPA JCA Deployment

Kodo can deploy Kodo JPA through the Java Connector Architecture (JCA) in any JCA-compliant application server that supports JDK 1.5 (all EJB 3 implementations require JDK 1.5). We present the deployment steps for the most common servers below.

Be sure to have your BEA license file available to the system classpath as detailed in [Section 2.2, “License Configuration” \[393\]](#)

8.1.3.1. WebLogic 9

First, ensure that your JDBC driver is in your system classpath. In addition, you will be adding the Kodo and specification API jars to the system classpath. You can accomplish this by editing `startWebLogic.sh/.cmd`.

Warning

Currently WebLogic ships with an old version of the EJB 3 libraries. Be sure to put `kodo.jar` in the *beginning* of the CLASSPATH.

The next step is to deploy `kodo-persistence.rar` from the `jca/persistence` directory of your Kodo installation. Copy this file to the `autodeploy` directory of your domain.

We will now extract `META-INF/ra.xml` and `META-INF/weblogic-ra.xml` to edit our configuration:

```
jar xvf kodo-ejb.rar META-INF/ra.xml META-INF/weblogic-ra.xml
```

Now you should configure Kodo JCA by editing `META-INF/ra.xml` substituting `config-property-value` stanzas with your own values. You can comment out properties (`config-property` stanzas) which you are not using or you can leave them at their default settings. Edit `META-INF/weblogic-ra.xml` to configure the JNDI location to which you want Kodo to be bound.

Now we can re-jar the manifest files back into the RAR file.

```
jar uvf kodo-ejb.rar META-INF/ra.xml META-INF/weblogic-ra.xml
rm META-INF/ra.xml META-INF/weblogic-ra.xml
rmdir META-INF
```

Now you can start WebLogic and WebLogic should deploy Kodo for you. If you have installed Kodo correctly, at this point, one should be able to see Kodo bound to the JNDI location which you specified earlier.

8.1.3.2. JBoss 4.x

First you must add the JPA specification jar, `jpa.jar`, which is found in the `lib` directory of the distribution, to the server library directory: e.g. `jboss-4.0.3/server/default/lib`). In addition, you should also place the appropriate JDBC driver jar into that same directory.

Note

JBoss currently ships with a stale version of the JPA specification jar. You should remove the old copy of this jar, `ejb3-persistence.jar`, from the server lib directory.

The `jca/persistence` directory of the Kodo distribution includes `kodo-persistence-jboss40-ds.xml`. This file should be edited to reflect your configuration, most notably connection values. Enter a JNDI name for Kodo to bind to (the default is `kodo-persistence`). Stop JBoss. Copy `kodo-persistence.rar` and `kodo-ejb-jboss40-ds.xml` to the deploy directory of your JBoss server installation (e.g. `jboss-4.0.3/server/default/deploy`). Once you have done this, you can restart JBoss to deploy Kodo.

To verify your installation, watch the console output for exceptions. In addition, you can check to see if Kodo was bound to JNDI. Open up your jmx-console (<http://yourhost:yourport/jmx-console>) and select the JNDIView service. If Kodo was installed correctly, you should see the Kodo connection factory bound at the JNDI name you specified. You have now installed Kodo JPA JCA.

8.1.4. Kodo JDO JCA Deployment

Kodo can deploy through the Java Connector Architecture (JCA) in any JCA-compliant application server. We present the deployment steps for the most common servers below.

Be sure to have your BEA license file available to the system classpath as detailed in [Section 2.2, “License Configuration”](#) [393]

8.1.4.1. WebLogic 6.1 to 7.x

Installation of Kodo into WebLogic requires 3 steps. First ensure that Kodo jars, the specification jars, and the appropriate JDBC driver are in the system classpath. In WebLogic 6.1.x, the classpath is set in the `startWebLogic.sh/cmd` in your domain directory (`$WL_HOME/config/mydomain`). In WebLogic 7, this file is the `startWLS.sh/cmd` file in the `$WL_HOME/server/bin` directory. Make sure to also add the JDO base jar (`jdo.jar`) to the classpath so that your JDO classes can be loaded. While this jar can be placed inside an ear file, putting it in the system classpath will reduce class resolution conflicts.

The `kodo-jdo.rar` file must then either be copied to the applications directory of your domain, or uploaded through the web admin interface. To upload using the web admin console, select `mydomain/Deployments/Connectors` in the left navigation bar and then select "Install a new Connector Component." Browse to `kodo-jdo.rar` and upload it to the server.

You should see `kodo-jdo` listed now in the `Connectors` folder in the navigation pane. Select it and select `Edit Connector Descriptor`. Under `RA`, expand `Config Properties` in the left pane and enter the appropriate values for each property. *Be sure to select `Apply` for every property.* In addition, you should provide a JNDI name for Kodo by selecting `WebLogic RA` from the navigation panel, entering an appropriate JNDI name, and selecting `Apply`. When you are done configuring Kodo, select the root (`kodo-jdo.rar`) of the navigation pane. Select `Persist` to save your configuration properties and propagate them to the active domain configuration.

You should see `WebLogic` attempt to deploy Kodo in the system console. When it is done, return to the main admin web console. Ensure that Kodo is deployed to your server by selecting `Targets` and adding your server to the chosen area. Kodo should now be deployed to your application server.

To verify the installation, you can view the JNDI tree for your server. Select the server from the admin navigation panel, right click on it, and select `View JNDI Tree` from the context menu. You should now see Kodo at the JNDI name you provided. You have now installed Kodo JDO JCA.

Note

When using WebLogic 7, you may get invalid DTD exceptions when loading JDO metadata files, due to a problem with loading resources that are in jar files inside a resource archive. You can work around these problems by specifying a non-public DTD in your metadata files, like so:

```
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_2_0.dtd">
```

8.1.4.2. WebLogic 8.1

Note

In its current version (8.1.0), there are a number of issues with classloaders in WebLogic's handling of `EAR` and `RAR` files. These instructions are aggressive in resolving potential issues which may no longer be issues in later releases of WebLogic.

First, ensure that your JDBC driver is in your system classpath. In addition, you will be adding the Kodo and specification API jars to the system classpath. You can accomplish this by editing `startWebLogic.sh/.cmd`.

The next step is to deploy `kodo-jdo.rar` from the `jca/jdo` directory of your Kodo installation. Create a directory named `kodo-jdo.rar` in the applications directory of your domain. `Un-jar kodo-jdo.rar` into this new directory (without copying `kodo-jdo.rar` itself).

Now you should configure Kodo JCA by editing `META-INF/ra.xml` substituting `config-property-value` stanzas with your own values. You can comment out properties (`config-property` stanzas) which you are not using or leaving at default settings. Edit `META-INF/weblogic-ra.xml` to configure the JNDI location to which you want Kodo to be bound.

Now you can start WebLogic and use the console to deploy Kodo JCA. Browse to your WebLogic admin port (`http://yourhost:7001/console`) and browse to the `Connectors (Deployments -> Connector Modules)` section and select `Deploy a new Connector`. Browse to and select `kodo-jdo.rar` and select `Target Modules` to ensure that Kodo is accessible to the proper servers.

If you have installed Kodo correctly, at this point, one should be able to see Kodo bound to the JNDI location which you spe-

cified earlier.

8.1.4.3. WebLogic 9

First, ensure that your JDBC driver is in your system classpath. In addition, you will be adding the Kodo and specification API jars to the system classpath. You can accomplish this by editing `startWebLogic.sh/.cmd`.

Warning

Currently WebLogic ships with an old version of the EJB 3 libraries. Be sure to put `kodo.jar` in the *beginning* of the CLASSPATH.

The next step is to deploy the RAR appropriate to the default specification you want to use. For example, to default to JPA, copy `kodo-ejb.rar` from the `jca/ejb` directory to the `autodeploy` directory of your domain.

We will now extract `META-INF/ra.xml` and `META-INF/weblogic-ra.xml` to edit our configuration:

```
jar xvf kodo-ejb.rar META-INF/ra.xml META-INF/weblogic-ra.xml
```

Now you should configure Kodo JCA by editing `META-INF/ra.xml` substituting `config-property-value` stanzas with your own values. You can comment out properties (`config-property` stanzas) which you are not using or leaving at default settings. Edit `META-INF/weblogic-ra.xml` to configure the JNDI location to which you want Kodo to be bound.

Now we can re-jar the manifest files back into the RAR file.

```
jar uvf kodo-ejb.rar META-INF/ra.xml META-INF/weblogic-ra.xml
rm META-INF/ra.xml META-INF/weblogic-ra.xml
rmdir META-INF
```

Now you can start WebLogic and WebLogic should deploy Kodo for you. If you have installed Kodo correctly, at this point, one should be able to see Kodo bound to the JNDI location which you specified earlier.

8.1.4.4. JBoss 3.0

The `jca/jdo` directory of the Kodo distribution includes `kodo-jdo-jboss30-service.xml`. This file should be edited to reflect your configuration, most notably connection values. Enter a JNDI name for Kodo to bind to (the default is `kodo-jdo`). Stop JBoss. Copy `kodo-jdo.rar` and `kodo-jdo-jboss30-service.xml` to the `deploy` directory of your JBoss server installation (e.g. `jboss-3.0.6/server/default/deploy`). Then copy the `jdo.jar` file to the `lib` directory of the JBoss server installation (e.g., `jboss-3.0.6/server/lib/`), so that the Kodo and the common specification APIs are available globally in the JBoss installation. In addition, you should also place the appropriate JDBC driver jar in the `lib` directory of your JBoss installation (e.g. `jboss-3.0.6/lib`).

To verify your installation, watch the console output for exceptions. In addition, you can check to see if Kodo was bound to JNDI. Open up your `jmx-console` (`http://yourhost:yourport/jmx-console`) and select the `JNDIView` service. If Kodo was installed correctly, you should see the Kodo connection factory bound at the JNDI name you specified. You have now installed Kodo JDO JCA.

8.1.4.5. JBoss 3.2

Installing in JBoss 3.2 is very similar to JBoss 3.0. Instead of editing and deploying `kodo-jdo-jboss30-service.xml`,

configuration is controlled by `kodo-jdo-jboss32-ds.xml`, also in the `jca/jdo` directory. Again, configuration involves supplying a JNDI name to bind Kodo, and setting up configuration values. These values are simple XML elements with the configuration property name as element name. `kodo-jdo-jboss32-ds.xml` and `kodo-jdo.rar` should be deployed to the deploy directory of JBoss. The installation is otherwise the same as JBoss 3.0.

8.1.4.6. JBoss 4.x

Installing in JBoss 4.x is very similar to JBoss 3.0 and 3.2. Instead of editing and deploying `kodo-jdo-jboss30-service.xml`, configuration is controlled by `kodo-jdo-jboss40-ds.xml`, also in the `jca/jdo` directory. Again, configuration involves supplying a JNDI name to bind Kodo, and setting up configuration values. These values are simple XML elements with the configuration property name as element name. `kodo-jdo-jboss4-ds.xml` and `kodo-jdo.rar` should be deployed to the deploy directory of JBoss. The installation is otherwise the same as JBoss 3.0.

8.1.4.7. WebSphere 5

WebSphere installation is easiest through the web admin interface. Open the admin console either by the `Start` menu item (in Windows), or by manually navigating to the admin port and URL appropriate to your installation. Select `Resources / Resource Adapters` from the left navigation panel. Select `Install Rar` on the list page. On the following screen upload `kodo-jdo.rar` to the server. On the `New` page, enter a name for the new Kodo installation such as `Kodo JDO JCA` and select `Ok`.

You should now be back to the `Resource Adapters` list page. Select the name of the Kodo installation you provided. Click on the link marked `J2C Connection Factories`. This is where you can configure a particular instance of Kodo's JCA implementation. Select `New` and you will be brought to a configuration page. *Be sure to fill in property values for Name and JNDI Name*. Select `Apply`.

After the page refreshes, select the `Custom Properties` link at the bottom of the page. On the `Custom Properties` page, you can enter in your connection and other configuration properties as necessary.

When you are done providing configuration values, you will want to save your changes back to the system configuration. Select the `Save` link on the page or the `Save` link in the menu bar. You have now installed Kodo JDO JCA.

8.1.4.8. SunONE Application Server 7 / Sun Java Enterprise Server 8-8.1

Installation in SunONE / Sun JES application server requires first providing JDO the proper permissions. This is accomplished by editing the `config/server.policy` for the server you are dealing with. Edit the file to include the following line into a `grant { } stanza`.

```
permission javax.jdo.spi.JDOPermission ".*";
```

Now restart the application server.

SunONE / Sun JES requires a Sun-specific deployment file in addition to the generic `ra.xml`. Edit the `sun-ra.xml` file provided in the `jca/jdo` directory of your Kodo installation by setting `jndi-name` attribute to the JNDI name of your choice. Then add `<property>` elements that correspond to the `<config-property-name>` in `ra.xml` to configure connection info and other configuration elements. Now update `kodo-jdo.rar` to include `sun-ra.xml` in the `META-INF` virtual directory in the archive:

```
mkdir META-INF
copy sun-ra.xml META-INF
jar -uvf kodo-jdo.rar META-INF/sun-ra.xml
```

Browse to the web admin console of SunONE / Sun JES in your browser. Select your server under `App Server Instances`

and expand to `Applications/Connector Modules`. Select `Deploy` and upload your new `kodo-jdo.rar` file. Enter an application name, and click the `Select` button. Apply your changes by selecting the link in the top right.

Note

Unfortunately, SunONE / Sun JES sometimes may not accept its own configuration file format. If this is the case, you will see exceptions as if your configuration values had not been set at all. To bypass this problem, extract `kodo-jdo.rar` into a temporary directory. Edit `ra.xml` and provide your configuration values directly into the `<config-property-value>` elements into the proper `<config-property>` stanzas.

If you have installed Kodo correctly, you should see `kodo-jdo` listed in the `Connector Module`. You have now installed Kodo JDO JCA.

8.1.4.9. Macromedia JRun 4

JRun requires a JRun-specific deployment file in addition to the generic `ra.xml`. Edit the `jrun-ra.xml` file provided in the `jca/jdo` directory of your Kodo installation by setting `jndi-name` attribute to the JNDI name of your choice. Then add `<property>` elements that correspond to the `<config-property-name>` in `ra.xml` to configure connection info and other configuration elements. Now update `kodo-jdo.rar` to include `jrun-ra.xml` in the `META-INF` virtual directory in the archive:

```
mkdir META-INF
copy jrun-ra.xml META-INF
jar -uvf kodo-jdo.rar META-INF/jrun-ra.xml
```

Browse to the web admin console of JRun in your browser. Select your server in the servers tree and expand to `J2EE Components`. Under `Resource Adapters`, select `Add` and upload your new `kodo-jdo.rar` file. The Kodo resource adapter will now be deployed to the JNDI name that you specified in the `jrun-ra.xml` file.

Note

JRun does not provide any means of configuring a built-in `DataSource` that is not enlisted in a global transaction. This means that when configuring Kodo to use an external `DataSource` bound into JRun, you must also configure Kodo's `Connection2URL`, `Connection2DriverName`, `Connection2UserName`, and `Connection2Password` properties in order to provide Kodo with a nontransactional connection.

8.1.4.10. Borland Enterprise Server 5.2 - 6.0

Note

Borland includes two different `TransactionManagers` for controlling the J2EE environment depending on your configuration. Kodo supports the standard instance on all versions of Borland. However, to use the two phase commit `TransactionManager`, labeled as the OTS service, you must use version 6.0 or higher. This is needed to support true XA transactions (prior versions had limited Java availability).

Due to classloader issues, as well as configuration of the RAR itself, some basic expertise in `jar` and `unjarring` is required to install Kodo into BES. Otherwise, we recommend deploying Kodo, then switching to single classpath for the entire system through the Admin tool (which prevents hot deploy) to ease classpath conflict issues. First extract `kodo-jdo.rar` to a temporary directory:

```
mkdir tmp
cd tmp
jar -xvf ../kodo-jdo.rar
```

From there, remove/move some jars that will cause class conflict issues with either your application or BES. Next, add the JDO specification API jars and your JDBC driver jars into the system classpath (e.g. `var/servers/your_server/partitions/lib/system`).

You must configure the RAR by editing the `ra-borland.xml` file included in the `jca/jdo` directory of your Kodo installation. Move this file into the `META-INF` directory of the expanded RAR file. Refer to the DTD and Borland's documentation on advanced features of the deployment descriptor, including deployment and security options.

With this completed, you can re-jar the expanded contents into a new `kodo-bes.rar` file to be deployed using `iastool` or the console interface. Before deploying, first *enable Visiconnect* on the partition using the console or the command-line utilities. This will activate JCA support in the application server. Then restart BES so that Visiconnect can activate and so that `jdo.jar` is added to the runtime classpath the runtime classpath. When deploying, stubs and verification do not need to be processed on the file and may simplify the deployment process.

```
tmp> jar -cvf kodo-bes.rar *
```

After a restart, Kodo should now be deployed to BES. You should be able to find Kodo at the JNDI location of `serial://kodo-jdo` in your application as well as be visible in the JNDI viewer in the console. You have now installed Kodo JDO JCA.

8.1.5. Non-JCA Application Server Deployment

For application servers that do not support JCA as a means of deploying resource adapters, Kodo can be deployed manually by writing some code to configure and bind an instance of the `PersistenceManagerFactory` into JNDI.

The mechanism by which you do this is up to you and will be dependent on functionality that is specific to your application server. The most common way is to create a startup class according to your application server's documentation that will create a factory and then bind it in JNDI. For example, in WebLogic you could write a startup class as follows:

Example 8.1. Binding a Factory into JNDI via a WebLogic Startup Class

JPA:

```
import java.util.*;
import javax.naming.*;

import weblogic.common.T3ServicesDef;
import weblogic.common.T3StartupDef;
import weblogic.jndi.WLContext;

/**
 * This startup class creates and binds an instance of a
 * Kodo EntityManagerFactory into JNDI.
 */
public class StartKodo
    implements T3StartupDef
{
    private static final String EMP_JNDI_NAME = "my.jndi.name";

    private T3ServicesDef services;

    public void setServices (T3ServicesDef services)
    {
        this.services = services;
    }

    public String startup (String name, Hashtable args)
        throws Exception
    {
        String jndi = (String) args.get ("jndiname");
        if (jndi == null || jndi.length () == 0)
            jndi = PMF_JNDI_NAME;

        EntityManagerFactory factory = Persistence.createEntityManagerFactory
            ("name of entity manager in META-INF/persistence.xml");
```

```

        Hashtable icprops = new Hashtable ();
        icprops.put (WLContext.REPLICATE_BINDINGS, "false");
        InitialContext ic = new InitialContext (icprops);
        ic.bind (jndi, factory);

        // return a message for logging
        return "Bound EntityManagerFactory to " + jndi;
    }
}

```

JDO:

```

import java.util.*;
import javax.naming.*;

import weblogic.common.T3ServicesDef;
import weblogic.common.T3StartupDef;
import weblogic.jndi.WLContext;

/**
 * This startup class creates and binds an instance of a
 * Kodo PersistenceManagerFactory into JNDI.
 */
public class StartKodo
    implements T3StartupDef
{
    private static final String PMF_JNDI_NAME = "my.jndi.name";
    private static final String PMF_PROPERTY =
        "javax.jdo.PersistenceManagerFactoryClass";
    private static final String PMF_CLASS_NAME =
        "kodo.jdo.PersistenceManagerFactoryImpl";

    private T3ServicesDef services;

    public void setServices (T3ServicesDef services)
    {
        this.services = services;
    }

    public String startup (String name, Hashtable args)
        throws Exception
    {
        String jndi = (String) args.get ("jndiname");
        if (jndi == null || jndi.length () == 0)
            jndi = PMF_JNDI_NAME;

        Properties props = new Properties ();
        props.setProperty (PMF_PROPERTY, PMF_CLASS_NAME);

        // you could set additional properties here; otherwise, the defaults
        // from kodo.properties will be used (if the file exists)

        PersistenceManagerFactory factory = JDOHelper.
            getPersistenceManagerFactory (props);

        Hashtable icprops = new Hashtable ();
        icprops.put (WLContext.REPLICATE_BINDINGS, "false");
        InitialContext ic = new InitialContext (icprops);
        ic.bind (jndi, factory);

        // return a message for logging
        return "Bound PersistenceManagerFactory to " + jndi;
    }
}

```

Applications that utilize Kodo can then obtain a handle to the factory as follows:

Example 8.2. Looking up the Factory in JNDI

JPA:

```
EntityManagerFactory factory = (EntityManagerFactory)
```

```
PortableRemoteObject.narrow (EntityManagerFactory.class,
    new InitialContext ().lookup ("java:/MyKodoJNDIName"));
EntityManager em = factory.getEntityManager ();
```

JDO:

```
PersistenceManagerFactory factory = (PersistenceManagerFactory)
    PortableRemoteObject.narrow (PersistenceManagerFactory.class,
        new InitialContext ().lookup ("java:/MyKodoJNDIName"));
PersistenceManager pm = factory.getPersistenceManager ();
```

8.2. Integrating with the Transaction Manager

Kodo `EntityManager`s and `PersistenceManager`s have the ability to automatically synchronize their transactions with an external transaction manager. Whether or not `EntityManager`s and `PersistenceManager`s from a given factory exhibit this behavior by default depends on the `kodo.TransactionMode` configuration property. The property can take the following values:

- `local`: Perform transaction operations locally.
- `managed`: Integrate with the application server's managed global transactions.

You can override the global transaction mode setting when you obtain an `EntityManager` using the `KodoEntityManagerFactory`'s `createEntityManager(PersistenceContextType ctype, boolean managed, int connRetainMode)` method.

You can override the global transaction mode setting when you obtain a `PersistenceManager` using the `KodoPersistenceManagerFactory`'s `getPersistenceManager(boolean managed, int connRetainMode)` method.

In order to use global transactions, Kodo must be able to access the application server's `javax.transaction.TransactionManager`. Kodo can automatically discover the transaction manager for most major application servers. Occasionally, however, you might have to point Kodo to the transaction manager for an unrecognized or non-standard application server setup. This is accomplished through the `kodo.ManagedRuntime` configuration property. This property describes a `kodo.ee.ManagedRuntime` implementation to use for transaction manager discovery. You can specify your own implementation, or use one of the built-ins:

- `auto`: This is the default. It is an alias for the `kodo.eeAutomaticManagedRuntime` class. This managed runtime is able to automatically integrate with several common application servers.
- `invocation`: An alias for the `kodo.ee.InvocationManagedRuntime` class. You can configure this runtime to invoke any static method in order to obtain the appserver's transaction manager.
- `jndi`: An alias for the `kodo.ee.JNDIManagedRuntime` class. You can configure this runtime to look up the transaction manager at any JNDI location.

See the Javadoc for each class for details on the bean properties you can pass to these plugins in your configuration string.

Important

This feature requires **Kodo Enterprise Edition**.

Example 8.3. Configuring Transaction Manager Integration

JPA XML format:

```
<property name="kodo.TransactionMode" value="managed" />
<property name="kodo.ManagedRuntime" value="jndi(TransactionManagerName=java:/TransactionManager)" />
```

JDO properties format:

```
kodo.TransactionMode: managed
kodo.ManagedRuntime: jndi(TransactionManagerName=java:/TransactionManager)
```

Note that even when Kodo is using managed transaction, you can control transactions through the specification local transaction APIs if you wish. Kodo will propagate your transaction calls to the global transaction.

8.3. XA Transactions

The X/Open Distributed Transaction Processing (X/Open DTP) model, designed by **Open Group** (a vendor consortium), defines a standard communication architecture that provides the following:

- Concurrent execution of applications on shared resources.
- Coordination of transactions across applications.
- Components, interfaces, and protocols that define the architecture and provide portability of applications.
- Atomicity of transaction systems.
- Single-thread control and sequential function-calling.

The X/Open DTP XA standard defines the application programming interfaces that a resource manager uses to communicate with a transaction manager. The XA interfaces enable resource managers to join transactions, to perform two-phase commit, and to recover in-doubt transactions following a failure.

8.3.1. Using Kodo with XA Transactions

Kodo supports XA-compliant transactions when used in a properly configured managed environment. The following components are required:

- A managed environment that provides an XA compliant transaction manager. Examples of this are application servers such as JBoss and WebLogic.
- Instances of a `javax.sql.XADataSource` for each of the `DataSources` that Kodo will use.

Important

This feature requires **Kodo Enterprise Edition**.

Given these components, setting up Kodo to participate in distributed transactions is a simple two-step process:

1. Integrate Kodo with your application server's transaction manager, as detailed in **Section 8.2, “Integrating with the Transaction Manager”** [535].
2. Point Kodo at an enlisted `XADataSource`, and configure a second non-enlisted data source. See **Section 4.2.1, “Managed and XA DataSources”** [426].

Chapter 9. Runtime Extensions

This chapter describes Kodo extensions to the standard JPA and JDO interfaces, and outlines some additional features of the Kodo runtime.

9.1. Architecture

Internally, Kodo does not adhere to any persistence specification. The Kodo kernel has its own set of APIs and components. Specifications like JPA and JDO are simply different "personalities" that can Kodo's native kernel can adopt.

As a Kodo JPA/JDO user, you will not normally see beneath Kodo's JPA and JDO personalities. Kodo allows you to access its feature set without leaving the comfort of JPA or JDO. Where Kodo goes beyond standard functionality, we have crafted JPA-specific and JDO-specific APIs to each Kodo extension for as seamless an experience as possible.

When writing Kodo plugins or otherwise extending the Kodo runtime, however, you will use Kodo's native APIs. So that you won't feel lost, the list below associates each specification interface with its backing native Kodo component:

- `javax.persistence.EntityManagerFactory`: `kodo.kernel.BrokerFactory`
- `javax.jdo.PersistenceManagerFactory`: `kodo.kernel.BrokerFactory`
- `javax.persistence.EntityManager`: `kodo.kernel.Broker`
- `javax.jdo.PersistenceManager`: `kodo.kernel.Broker`
- `javax.persistence.Query`: `kodo.kernel.Query`
- `javax.jdo.Query`: `kodo.kernel.Query`
- `kodo.persistence.Extent`: `kodo.kernel.Extent`
- `javax.jdo.Extent`: `kodo.kernel.Extent`
- `kodo.persistence.StoreCache`: `kodo.datacache.DataCache`
- `javax.jdo.datastore.DataStoreCache`: `kodo.datacache.DataCache`
- `kodo.persistence.QueryResultCache`: `kodo.datacache.QueryCache`
- `kodo.jdo.QueryResultCache`: `kodo.datacache.QueryCache`
- `kodo.persistence.FetchPlan`: `kodo.kernel.FetchConfiguration`
- `javax.jdo.FetchPlan`: `kodo.kernel.FetchConfiguration`
- `kodo.persistence.Generator`: `kodo.kernel.Seq`
- `javax.jdo.datastore.Sequence`: `kodo.kernel.Seq`

The `kodo.persistence.KodoPersistence` helper allows you to convert between `EntityManagerFactories` and `BrokerFactories`, `EntityManagers` and `Brokers`.

The `kodo.jdo.KodoJDOHelper` allows you to convert between `PersistenceManagerFactories` and `BrokerFactories`, `PersistenceManagers` and `Brokers`.

As a Kodo JPA/JDO user, you can use these methods to move from an JPA to a JDO persistence API (or vice versa) at any time.

You can even switch back and forth within the same persistence context, as the following example illustrates:

Example 9.1. Switching APIs within a Persistence Context

```
import kodo.persistence.*;
...
EntityManager em = ...;
em.getTransaction().begin();

Magazine mag = em.find(Magazine.class, magId);
setHigherSellers(mag);

em.getTransaction().commit();
em.close();

...
private void setHigherSellers(Magazine mag)
{
    // or we could get the EM using KodoPersistence, convert it to a Broker,
    // and convert the Broker to a PM using KodoJDOHelper. this is easier
    PersistenceManager pm = JDOHelper.getPersistenceManager(mag);
    Query q = pm.newQuery(Magazine.class, "sales > :s");
    List results = (List) q.execute(mag.getSales());
    mag.setHigherSellingMagazines(new HashSet(results));
}
```

9.1.1. Broker Customization

Some advanced users may want to add capabilities to Kodo's internal `kodo.kernel.BrokerImpl`. You can configure Kodo to use a custom subclass of `BrokerImpl` through the `kodo.BrokerImpl` configuration property. Set this property to the full class name of your custom subclass.

As a **plugin string**, you can also use this property to configure the `BrokerImpl` with the following properties:

- `EvictFromDataCache`: When evicting an object through the `KodoEntityManager.evict` or `PersistenceManager.evict` methods, whether to also evict it from the Kodo's **data cache**. Defaults to `false`.

Example 9.2. Evict from Data Cache

JPA XML format:

```
<property name="kodo.BrokerImpl" value="EvictFromDataCache=true"/>
```

JDO properties format:

```
kodo.BrokerImpl: EvictFromDataCache=true
```

9.2. JPA Extensions

The following sections outline the runtime interfaces you can use to access Kodo-specific functionality from JPA. Each interface contains services and convenience methods missing from the JPA specification. Kodo strives to use the same naming conventions and API patterns as standard JPA methods in all extensions, so that Kodo JDO APIs feel as much as possible like standard JPA.

You may have noticed the examples throughout this document using the `KodoPersistence.cast` methods to cast from standard JPA interfaces to Kodo extended interfaces. This is the recommended practice. Some application server vendors may proxy Kodo's JPA implementation, preventing a straight cast. KodoPersistence's `cast` methods work around these proxies.

```
public static KodoEntityManagerFactory cast (EntityManagerFactory emf);
public static KodoEntityManager cast (EntityManager em);
public static KodoQuery cast (Query q);
```

We provide additional information on the `KodoPersistence` helper [below](#).

9.2.1. KodoEntityManagerFactory

The `kodo.persistence.KodoEntityManagerFactory` interface extends the basic `javax.persistence.EntityManagerFactory` with Kodo-specific features. The `KodoEntityManagerFactory` offers APIs to obtain managed and unmanaged `EntityManagers` from the same factory, to access the Kodo data and query caches, and to perform other Kodo-specific operations. See the [interface Javadoc](#) for details.

9.2.2. KodoEntityManager

All Kodo `EntityManagers` implement the `kodo.persistence.KodoEntityManager` interface. This interface extends the standard `javax.persistence.EntityManager`. Just as the standard `EntityManager` is the primary window into JPA services, the `KodoEntityManager` is the primary window from JPA into Kodo-specific functionality. We strongly encourage you to investigate the API extensions this interface contains.

9.2.3. KodoQuery

Kodo extends JPA's standard query functionality with the `kodo.persistence.KodoQuery` interface. See its [Javadoc](#) for details on the convenience methods it provides.

9.2.4. Extent

An `Extent` is a logical view of all persistent instances of a given entity class, possibly including subclasses. Kodo adds the `kodo.persistence.Extent` class to the set of Java Persistence APIs. The following code illustrates iterating over all instances of the `Magazine` entity, without subclasses:

Example 9.3. Using a JPA Extent

```
import kodo.persistence.*;
...
KodoEntityManager kem = KodoPersistence.cast (em);
Extent<Magazine> mags = kem.getExtent (Magazine.class, false);
for (Magazine m : mags)
    processMagazine (m);
```

9.2.5. StoreCache

In addition to the `EntityManager` object cache mandated by the JPA specification, Kodo includes a flexible datastore-level cache. You can access this cache from your JPA code using the `kodo.persistence.StoreCache` facade. [Section 10.1, “Data Cache” \[560\]](#) has detailed information on Kodo's data caching system, including the `StoreCache` facade.

9.2.6. QueryResultCache

Kodo can cache query results as well as persistent object data. The `kodo.persistence.QueryResultCache` is an JPA-flavored facade to Kodo's internal query cache. See [Section 10.1.3, “Query Cache” \[560\]](#) for details on query caching in Kodo.

9.2.7. FetchPlan

Many of the aforementioned Kodo interfaces give you access to a `kodo.persistence.FetchPlan` instance. The `FetchPlan` allows you to exercise some control over how objects are fetched from the datastore, including **large result set support**, **custom fetch groups**, and **lock levels**.

Kodo goes one step further, extending `FetchPlan` with `kodo.persistence.jdbc.JDBCFetchPlan` to add additional JDBC-specific tuning methods. Unless you have customized Kodo to use a non-relational back-end (see [Section 9.9, “Non-Relational Stores” \[559\]](#)) all `FetchPlans` in Kodo implement `JDBCFetchPlan`, so feel free to cast to this interface.

Fetch plans pass on from parent components to child components. The `EntityManagerFactory` settings (via your configuration properties) for things like the fetch size, result set type, and custom fetch groups are passed on to the fetch plan of the `EntityManagers` it produces. The settings of each `EntityManager`, in turn, are passed on to each `Query` and `Extent` it returns. Note that the opposite, however, is not true. Modifying the fetch plan of a `Query` or `Extent` does not affect the `EntityManager`'s configuration. Likewise, modifying an `EntityManager`'s configuration does not affect the `EntityManagerFactory`.

[Section 5.6, “Fetch Groups” \[469\]](#) includes examples using `FetchPlans`.

9.2.8. KodoPersistence

`kodo.persistence.KodoPersistence` is a static helper class that adds Kodo-specific utility methods to `javax.persistence.Persistence`.

9.3. JDO API Extensions

The following sections outline the runtime interfaces you can use to access Kodo-specific functionality from JDO. Each interface contains services and convenience methods missing from the JDO specification. Kodo strives to use the same naming conventions and API patterns as standard JDO methods in all extensions, so that Kodo JDO APIs feel as much as possible like standard JDO APIs.

9.3.1. KodoPersistenceManagerFactory

The `kodo.jdo.KodoPersistenceManagerFactory` interface extends the basic `javax.jdo.PersistenceManagerFactory` with Kodo-specific features. The `KodoPersistenceManagerFactory` offers APIs to obtain managed and unmanaged `PersistenceManagers` from the same factory, to access the query cache, and to perform other Kodo-specific operations. See the [interface Javadoc](#) for details.

9.3.2. KodoPersistenceManager

All Kodo `PersistenceManagers` implement the `kodo.jdo.KodoPersistenceManager` interface. This interface extends the standard `javax.jdo.PersistenceManager`. Just as the standard `PersistenceManager` is the primary window into JDO runtime services, the `KodoPersistenceManager` is the primary window from JDO into Kodo-specific func-

tionality. We strongly encourage you to investigate the API extensions this interface contains.

9.3.3. KodoQuery

Kodo extends JDO's standard query functionality with the `kodo.jdo.KodoQuery` interface. See its **Javadoc** for details on the convenience methods it provides.

9.3.4. KodoExtent

The `kodo.jdo.KodoExtent` offers convenience methods not found in standard `javax.jdo.Extents`.

9.3.5. KodoDataStoreCache

Kodo expands JDO's standard `javax.jdo.datastore.DataStoreCache` interface with the `kodo.jdo.KodoDataStoreCache`. [Section 10.1, “Data Cache” \[560\]](#) has detailed information on Kodo's data caching system, including the `KodoDataStoreCache`.

9.3.6. QueryResultCache

Kodo includes a JDO-flavored facade to its internal query cache. The `kodo.jdo.QueryResultCache` includes APIs much like those in the `DataStoreCache`, but acting on queries rather than persistent objects. See [Section 10.1.3, “Query Cache” \[?\]](#) for details on query caching in Kodo.

9.3.7. KodoFetchPlan

The `kodo.jdo.KodoFetchPlan` adds additional options to the standard `kodo.jdo.FetchPlan` for fine-tuning data loading. Its APIs include **large result set support** and **lock level** control, among others.

Kodo goes one step further, extending `KodoFetchPlan` with `kodo.jdo.jdbc.JDBCFetchPlan` to add additional JDBC-specific tuning methods. Unless you have customized Kodo to use a non-relational back-end (see [Section 9.9, “Non-Relational Stores” \[559\]](#)) all JDO `FetchPlans` in Kodo implement `JDBCFetchPlan`, so feel free to cast to this interface.

9.3.8. KodoJDOHelper

The `kodo.jdo.KodoJDOHelper` is a static helper class that adds Kodo-specific utility methods to `javax.jdo.JDOHelper`.

9.4. Object Locking

Controlling how and when objects are locked is an important part of maximizing the performance of your application under load. This section describes Kodo's APIs for explicit locking, as well as its rules for implicit locking.

9.4.1. Configuring Default Locking

You can control Kodo's default transactional read and write lock levels through the `kodo.ReadLockLevel` and `kodo.WriteLockLevel` configuration properties. Each property accepts a value of `none`, `read`, `write`, or a number corresponding to a lock level defined by the **lock manager** in use. These properties apply only to non-optimistic transactions; during optimistic transactions, Kodo never locks objects by default.

You can control the default amount of time Kodo will wait when trying to obtain locks through the `kodo.LockTimeout` configuration property. Set this property to the number of milliseconds you are willing to wait for a lock before Kodo will throw an exception, or to `-1` for no limit. It defaults to `-1`.

Example 9.4. Setting Default Lock Levels

JPA XML format:

```
<property name="kodo.ReadLockLevel" value="none"/>
<property name="kodo.WriteLockLevel" value="write"/>
<property name="kodo.LockTimeout" value="30000"/>
```

JDO properties format:

```
kodo.ReadLockLevel: none
kodo.WriteLockLevel: write
kodo.LockTimeout: 30000
```

9.4.2. Configuring Lock Levels at Runtime

At runtime, you can override the default lock levels in JPA through the `kodo.persistence.FetchPlan`, or in JDO through the `kodo.jdo.KodoFetchPlan`. These interfaces are described above. At the beginning of each datastore transaction, Kodo initializes the `EntityManager` or `PersistenceManager`'s fetch plan with the default lock levels and timeouts described in the previous section. By changing the fetch plan's locking properties, you can control how objects loaded at different points in the transaction are locked. You can also use the fetch plan of an individual `Query` or `Extent` to apply your locking changes only to objects loaded through that `Query` or `Extent`.

JPA `FetchPlan`. A null `LockModeType` represents no locking:

```
public LockModeType getReadLockMode ();
public FetchPlan setReadLockMode (LockModeType mode);
public LockModeType getWriteLockMode ();
public FetchPlan setWriteLockMode (LockModeType mode);
long getLockTimeout ();
FetchPlan setLockTimeout (long timeout);
```

JDO `KodoFetchPlan`:

```
public int getReadLockLevel ();
public KodoFetchPlan setReadLockLevel (int level);
public int getWriteLockLevel ();
public KodoFetchPlan setWriteLockLevel (int level);
long getLockTimeout ();
KodoFetchPlan setLockTimeout (long timeout);
```

Controlling locking through these runtime APIs works even during optimistic transactions. At the end of the transaction, Kodo resets the fetch plan's lock levels to none. You cannot lock objects outside of a transaction.

Example 9.5. Setting Runtime Lock Levels

JPA:

```
import kodo.persistence.*;
...
EntityManager em = ...;
em.getTransaction ().begin ();

// load stock we know we're going to update at write lock mode
Query q = em.createQuery ("select s from Stock s where symbol = :s");
q.setParameter ("s", symbol);
KodoQuery kq = KodoPersistence.cast (q);
FetchPlan fetch = kq.getFetchPlan ();
fetch.setReadLockMode (LockModeType.WRITE);
fetch.setLockTimeout (3000); // 3 seconds
Stock stock = (Stock) q.getSingleResult ();

// load an object we don't need locked at none lock mode
fetch = (KodoPersistence.cast (em)).getFetchPlan ();
fetch.setReadLockMode (null);
Market market = em.find (Market.class, marketId);

stock.setPrice (market.calculatePrice (stock));
em.getTransaction ().commit ();
```

JDO:

```
import kodo.jdo.*;
...
PersistenceManager pm = ...;
pm.currentTransaction ().begin ();

// load stock we know we're going to update at write lock level
Query q = pm.newQuery (Stock.class, "symbol == :s");
q.setUnique (true);
KodoFetchPlan fetch = (KodoFetchPlan) q.getFetchPlan ();
fetch.setReadLockLevel (KodoFetchPlan.LOCK_WRITE);
fetch.setLockTimeout (3000); // 3 seconds
Stock stock = (Stock) q.execute (symbol);

// load an object we don't need locked at none lock level
fetch = (KodoFetchPlan) pm.getFetchPlan ();
fetch.setReadLockLevel (KodoFetchPlan.LOCK_NONE);
Market market = (Market) pm.getObjectById (marketId);

stock.setPrice (market.calculatePrice (stock));
pm.currentTransaction ().commit ();
```

9.4.3. Object Locking APIs

In addition to allowing you to control implicit locking levels, Kodo provides explicit APIs to lock objects and to retrieve their current lock level.

JPA:

```
public LockModeType KodoEntityManager.getLockMode (Object pc);
```

JDO:

```
public static int KodoJDOHelper.getLockLevel (Object pc);
```

Returns the level at which the given object is currently locked.

In addition to the standard `EntityManager.lock (Object, LockModeType)` method, the `KodoEntityManager` exposes the following methods to lock objects explicitly:

```
public void lock (Object pc);
public void lock (Object pc, LockModeType mode, long timeout);
public void lockAll (Object... pcs);
public void lockAll (Object... pcs, LockModeType mode, long timeout);
public void lockAll (Collection pcs);
public void lockAll (Collection pcs, LockModeType mode, long timeout);
```

The Kodo JDO `KodoPersistenceManager` exposes the following methods to lock objects explicitly:

```
public void lockPersistent (Object pc);
public void lockPersistent (Object pc, int level, long timeout);
public void lockPersistentAll (Object[] pcs);
public void lockPersistentAll (Object[] pcs, int level, long timeout);
public void lockPersistentAll (Collection pcs);
public void lockPersistentAll (Collection pcs, int level, long timeout);
```

Methods that do not take a lock level or timeout parameter default to the current fetch plan. The example below demonstrates these methods in action.

Example 9.6. Locking APIs

JPA:

```
import kodo.persistence.*;

// retrieve the lock level of an object
KodoEntityManager kem = KodoPersistence.cast (em);
Stock stock = ...;
LockModeType level = kem.getLockMode (stock);
if (level == KodoModeType.WRITE) ...

...

kem.setOptimistic (true);
kem.getTransaction ().begin ();

// override default of not locking during an opt trans to lock stock object
kem.lock (stock, LockModeType.WRITE, 1000);
stock.setPrice (market.calculatePrice (stock));

kem.getTransaction ().commit ();
```

JDO:

```
import kodo.jdo.*;

// retrieve the lock level of an object
Stock stock = ...;
int level = KodoJDOHelper.getLockLevel (stock);
if (level == KodoJDOHelper.LOCK_WRITE) ...
```

```

...
PersistenceManager pm = ...;
pm.currentTransaction().setOptimistic(true);
pm.currentTransaction().begin();

// override default of not locking during an opt trans to lock stock object
KodoPersistenceManager kpm = KodoJDOHelper.cast(pm);
kpm.lockPersistent(stock, KodoPersistenceManager.LOCK_WRITE, -1);
stock.setPrice(market.calculatePrice(stock));

pm.currentTransaction().commit();

```

9.4.4. Lock Manager

Kodo delegates the actual work of locking objects to the system's `kodo.kernel.LockManager`. This plugin is controlled by the `kodo.LockManager` configuration property. You can write your own lock manager, or use one of the bundled options:

- `pessimistic`: This is an alias for the `kodo.jdbc.kernel.PessimisticLockManager`, which uses `SELECT FOR UPDATE` statements (or the database's equivalent) to lock the database rows corresponding to locked objects. This lock manager does not distinguish between read locks and write locks; all locks are write locks.

The `pessimistic` `LockManager` can be configured to additionally perform the version checking and incrementing behavior of the version lock manager described below by setting its `VersionCheckOnReadLock` and `VersionUpdateOnWriteLock` properties:

JPA XML format:

```
<property name="kodo.LockManager" value="pessimistic(VersionCheckOnReadLock=true,VersionUpdateOnWriteLock=true)"/>
```

JDO properties format:

```
kodo.LockManager: pessimistic(VersionCheckOnReadLock=true,VersionUpdateOnWriteLock=true)
```

This is the default `kodo.LockManager` setting in JDO.

- `none`: An alias for the `kodo.kernel.NoneLockManager`, which does not perform any locking at all.
- `sjvm`: An alias for the `kodo.kernel.SingleJVMExclusiveLockManager`. This lock manager uses in-memory mutexes to obtain exclusive locks on object ids. It does not perform any database-level locking. Also, it does not distinguish between read and write locks; all locks are write locks.
- `version`: An alias for the `kodo.kernel.VersionLockManager`. This lock manager does not perform any exclusive locking, but instead ensures read consistency by verifying that the version of all read-locked instances is unchanged at the end of the transaction. Furthermore, a write lock will force an increment to the version at the end of the transaction, even if the object is not otherwise modified. This ensures read consistency with non-blocking behavior.

This is the default `kodo.LockManager` setting in JPA.

Note

In order for the `version` lock manager to prevent the dirty read phenomenon, the underlying data store's transaction isolation level must be set to the equivalent of "read committed" or higher.

Example 9.7. Disabling Locking

JPA XML format:

```
<property name="kodo.LockManager" value="none" />
```

JDO properties format:

```
kodo.LockManager: none
```

9.4.5. Rules for Locking Behavior

Advanced persistence concepts like lazy-loading and object uniquing create several locking corner-cases. The rules below outline Kodo's implicit locking behavior in these cases.

1. When an object's state is first read within a transaction, the object is locked at the fetch plan's current read lock level. Future reads of additional lazy state for the object will use the same read lock level, even if the fetch plan's level has changed.
2. When an object's state is first modified within a transaction, the object is locked at the write lock level in effect when the object was first read, even if the fetch plan's level has changed. If the object was not read previously, the current write lock level is used.
3. When objects are accessed through a persistent relation field, the related objects are loaded with the fetch plan's current lock levels, not the lock levels of the object owning the field.
4. Whenever an object is accessed within a transaction, the object is re-locked at the current read lock level. The current read and write lock levels become those that the object "remembers" according to rules one and two above.
5. If you lock an object explicitly through the APIs demonstrated above, it is re-locked at the specified level. This level also becomes both the read and write level that the object "remembers" according to rules one and two above.
6. When an object is already locked at a given lock level, re-locking at a lower level has no effect. Locks cannot be downgraded during a transaction.

9.4.6. Known Issues and Limitations

Due to performance concerns and database limitations, locking cannot be perfect. You should be aware of the issues outlined in this section, as they may affect your application.

- Typically, during optimistic transactions Kodo does not start an actual database transaction until you flush or the optimistic transaction commits. This allows for very long-lived transactions without consuming database resources. When using the default lock manager, however, Kodo must begin a database transaction whenever you decide to lock an object during an optimistic transaction. This is because the default lock manager uses database locks, and databases cannot lock rows without a transaction in progress. Kodo will log an INFO message to the `kodo.Runtime` logging channel when it begins a datastore transaction just to lock an object.
- In order to maintain reasonable performance levels when loading object state, Kodo can only guarantee that an object is locked at the proper lock level *after* the state has been retrieved from the database. This means that it is technically possible for another transaction to "sneak in" and modify the database record after Kodo retrieves the state, but before it locks the object. The only way to positively guarantee that the object is locked and has the most recent state to refresh the object after locking it.

When using the default lock manager, the case above can only occur when Kodo cannot issue the state-loading SELECT as a locking statement due to database limitations. For example, some databases cannot lock SELECTs that use joins. The default lock manager will log an INFO message to the `kodo.Runtime` logging channel whenever it cannot lock the initial SELECT due to database limitations. By paying attention to these log messages, you can see where you might consider using an object refresh to guarantee that you have the most recent state, or where you might rethink the way you load the state in question to circumvent the database limitations that prevent Kodo from issuing a locking SELECT in the first place.

9.5. Savepoints

Savepoints allow for fine grained control over the transactional behavior of your application. Kodo's savepoint API allow you to set intermediate rollback points in your transaction. You can then choose to rollback changes made only after a specific savepoint, then commit or continue making new changes in the transaction. This feature is useful for multi-stage transactions, such as editing a set of objects over several web pages or user screens. Savepoints also provide more flexibility to conditional transaction behavior, such as choosing to commit or rollback a portion of the transaction based on the results of the changes. This chapter describes how to use and configure Kodo savepoints.

9.5.1. Using Savepoints

Kodo's `KodoEntityManager` and `KodoPersistenceManager` have the following methods to control savepoint behavior. Note that the savepoints work in tandem with the current transaction. This means that savepoints require an open transaction, and that a rollback of the transaction will rollback all of the changes in the transaction regardless of any savepoints set.

```
void setSavepoint (String name);
void releaseSavepoint (String name);
void rollbackToSavepoint (String name);
```

To set a savepoint, simply call `setSavepoint`, passing in a symbolic savepoint name. This savepoint will define a point at which you can preserve the state of transactional objects for the duration of the current transaction.

Having set a named savepoint, you can rollback changes made after that point by calling `rollbackToSavepoint`. This method will keep the current transaction active, while restoring all transactional instances back to their saved state. Instances that were deleted after the save point will no longer be marked for deletion. Similarly, transient instances that were made persistent after the savepoint will become transient again. Savepoints made after this savepoint will be released and no longer valid, although you can still set new savepoints. Savepoints will also be cleared after the current transaction is committed or rolled back.

If a savepoint is no longer needed, you can release any resources such as in memory state and datastore resources by calling `releaseSavepoint`. This method should not be called for savepoints that have been released automatically through other means, such as commit of a transaction or rollback to a prior savepoint. While savepoints made after this savepoint will also be released, there are no other effects on the current transaction.

The following simple example illustrates setting, releasing, and rolling back to a savepoint.

Example 9.8. Using Savepoints

JPA:

```

import kodo.persistence.*;

...

KodoEntityManager kem = KodoPersistence.cast (em);
kem.getTransaction ().begin ();

Magazine mag = kem.find (Magazine.class, id);
mag.setPageCount (300);
kem.setSavepoint ("pages");

mag.setPrice (mag.getPageCount () * pricePerPage);
// we decide to release pages since price depends on pages.
kem.releaseSavepoint ("pages");
kem.setSavepoint ("price");

mag.setPrice (testPrice);
...

// we determine the test price is not good
kem.rollbackToSavepoint ("price");

// the price is now restored to mag.getPageCount () * pricePerPage
kem.getTransaction ().commit ();

```

JDO:

```

import kodo.jdo.*;

...

KodoPersistenceManager kpm = KodoJDOHelper.cast (pm);
kpm.currentTransaction ().begin ();

Magazine mag = (Magazine) kpm.getObjectById (id, true);
mag.setPageCount (300);
kpm.setSavepoint ("pages");

mag.setPrice (mag.getPageCount () * pricePerPage);
// we decide to release pages since price depends on pages.
kpm.releaseSavepoint ("pages");
kpm.setSavepoint ("price");

mag.setPrice (testPrice);
...

// we determine the test price is not good
kpm.rollbackToSavepoint ("price");

// the price is now restored to mag.getPageCount () * pricePerPage
kpm.currentTransaction ().commit ();

```

9.5.2. Configuring Savepoints

Kodo uses the `kodo.kernel.SavepointManager` plugin to handle perserving the savepoint state. Kodo includes the following `SavepointManager` plugins:

- `in-mem`: The default. This is an alias for the `kodo.kernel.InMemorySavepointManager`. This plugin stores all state, including field values, in memory. Due to this behavior, each set savepoint is designed for small to medium transactional object counts.

- `jdbc`: This is an alias for the `kodo.jdbc.kernel.JDBCSavepointManager`. This plugin requires JDBC 3 and `java.sql.Savepoint` support to operate. Note that this plugin implements savepoints by issuing a flush to the database.
- `oracle`: This is an alias for the `kodo.jdbc.sql.OracleSavepointManager`. This plugin operates similarly to the JDBC plugin; however, it uses Oracle-specific calls. This plugin requires using the Oracle JDBC driver and database, versions 9.2 or higher. Note that this plugin implements savepoints by issuing a flush to the database.

9.6. Query Language Extensions

JPQL and JDOQL are powerful, easy-to-use query languages, but you may occasionally find them limiting in some way. To circumvent the limitations of JDOQL and JPQL, Kodo provides extensions to these languages, and allows you extend them as well.

Warning

The JPQL parser in this release does not yet allow extensions. They will be made available to JPQL users in a future release.

9.6.1. Filter Extensions

Filter extensions are custom methods that you can use in your query filter, having, ordering, and result strings. Kodo provides some built-in filter extensions, and you can develop your own custom extensions as needed. You can optionally preface all filter extensions with `ext:` in your query string. For example, the following example uses a hypothetical `firstThreeChars` extension to search for cities whose name begins with the 3 characters 'H', 'a', 'r'.

Example 9.9. Basic Filter Extension

JPA:

```
Query q = em.createQuery ("select c from City c where c.name.ext:firstThreeChars () = 'Har'");
List results = q.getResultList ();
```

JDO:

```
Query q = pm.newQuery (City.class, "name.ext:firstThreeChars () == 'Har'");
List results = (List) q.execute ();
```

Note that it is perfectly OK to chain together extensions. For example, let's modify our search above to be case-insensitive using another hypothetical extension, `equalsIgnoreCase`:

Example 9.10. Chaining Filter Extensions

JPA:

```
Query query = em.createQuery ("select c from City c where "
+ "c.name.ext:firstThreeChars ().ext>equalsIgnoreCase ('Har')");
```

```
List results = q.getResultList ();
```

JDO:

```
Query q = pm.newQuery (City.class, "name.ext:firstThreeChars ().ext:equalsIgnoreCase ('Har')");
List results = (List) q.execute ();
```

Finally, when using filter extensions you must be aware that any SQL-specific extensions can only execute against the database, and cannot be used for in-memory queries (recall that Kodo executes queries in-memory when you supply a candidate collection rather than a class, or when you set the `IgnoreChanges` and `FlushBeforeQueries` properties to `false` and you execute a query within a transaction in which you've modified data that may affect the results).

9.6.1.1. Included Filter Extensions

Kodo includes two default filter extensions to enhance the power of your queries.

- `getColumn`: Places the proper alias for the given column name into the `SELECT` statement that is issued. This extension cannot be used for in-memory queries. When traversing relations, the column is assumed to be in the primary table of the related type.

JPQL:

```
select e from Employee e where e.company.address.ext:getColumn ('ID') = 5
```

To get a column of the candidate class in JDO, use `this` as the extension target, as shown in the second JDOQL example below.

JDOQL:

```
company.address.ext:getColumn ('ID') == 5
this.ext:getColumn ('LEGACY_DATA') == 'foo'
```

- `sql`: Embeds the given SQL argument into the `SELECT` statement. This extension cannot be used for in-memory queries.

JPQL:

```
select p from Product p where p.price < ext:sql (('SELECT AVG(PRICE) FROM PRODUCTS'))
```

JDOQL:

```
price < ext:sql (('SELECT AVG(PRICE) FROM PRODUCTS'))
```

9.6.1.2. Developing Custom Filter Extensions

You can write your own extensions by implementing the `kodo.jdbc.kernel.exps.JDBCFilterListener` interface. View the Javadoc documentation for details. Additionally, the source for all of Kodo's built-in query extensions is included in your Kodo download to get you started. The built-in extensions reside in the `src/kodo/kernel/exps` and `src/kodo/jdbc/kernel/exps` directories of your distribution.

9.6.1.3. Configuring Filter Extensions

There are two ways to register your custom filter extensions with Kodo:

- *Registration by properties:* You can register custom filter extensions by setting the `kodo.FilterListeners` configuration property to a comma-separated list of plugin strings (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing your extensions classes. Extensions registered in this fashion must have a public no-arg constructor. They must also be thread safe, because they will be shared across all queries.
- *Per-query registration:* You can register filter extensions for an individual Query through the `KodoQuery.addFilterListener` method. You might use per-query registration for very specific extensions that do not apply globally.

See the the `kodo.persistence.KodoQuery` and `kodo.jdo.KodoQuery` Javadoc for details.

9.6.2. Aggregate Extensions

Just as you can write your own filter methods, you can write your own query aggregates by implementing the `kodo.jdbc.kernel.exps.JDBCAggregateListener` interface. View the Javadoc documentation for details. When using your custom aggregates in result or having query clauses, you can optionally prefix the function name with `ext :` to identify it as an extension.

9.6.2.1. Configuring Query Aggregates

There are two ways to register your custom query aggregates with Kodo:

- *Registration by properties:* You can register custom query aggregates by setting the `kodo.AggregateListeners` configuration property to a comma-separated list of plugin strings (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing your aggregate implementation. Aggregates registered in this fashion must have a public no-arg constructor. They must also be thread safe, because they will be shared across all queries.
- *Per-query registration:* You can register query aggregates for an individual Query through the `KodoQuery.addAggregateListener` method. You might use per-query registration for very specific aggregates that do not apply globally.

See the the `kodo.persistence.KodoQuery` and `kodo.jdo.KodoQuery` Javadoc for details.

9.6.3. JDOQL Non-Distinct Results

The JDO specification mandates that an implementation must always filter out duplicate query results caused by the semantics of the underlying datastore. When querying a relational database, for example, Kodo must often issue `DISTINCT` SQL queries or even subselects to avoid including repeated data in your results.

Kodo allows you to explicitly request unfiltered results with the `nondistinct` keyword. If used, `nondistinct` must be the first word of your query's result clause. Like other JDOQL keywords, `nondistinct` can be expressed in either all-lower or all-upper case. [Chapter 11, Query \[232\]](#) covers the standard JDOQL language.

Example 9.11. Nondistinct JDOQL

```
Query q = pm.newQuery (Magazine.class, "select nondistinct this "
    + "where articles.contains (art) && art.author.lastName == 'Smith'");
```

9.6.4. JDOQL Subqueries

The JDO specification does not provide a simple way to embed one query within another. The best you can do is perform the "inner" query, and use its result(s) as a parameter to the "outer" query. Kodo JDO corrects this oversight with full support for JDOQL subqueries.

Important

This feature requires **Kodo Enterprise Edition**.

Kodo's JDOQL subqueries utilize the single-string JDOQL syntax, explained in [Section 11.9, "Single-String JDOQL" \[248\]](#) of the JDO Overview. Kodo makes one minor addition to the single-string format, requiring a logical alias for the subquery candidates. As you'll see below, this allows you to create *correlated* subqueries that include values from the parent query. The results of a subquery are supplied to the parent query as either a single value or a collection, depending on usage. Let's examine some examples of subqueries in action.

Example 9.12. Comparison to Subquery

In this example, we find all magazines that are tied for the highest cover price.

```
Query q = pm.newQuery (Magazine.class, "price == (select max(m.price) "
    + "from org.mag.Magazine m)");
List mags = (List) q.execute ();
```

Notice that the example surrounds its subquery in parentheses. This is required for all subqueries. Also, note the use of the `m` alias for subquery candidates. The candidate alias is also required. Without it, Kodo couldn't support correlated subqueries, like the one in the next example.

Example 9.13. Correlated Subquery

In this example, we find all the magazines tied for the highest price within their publisher. The subquery is *correlated* because it uses the `publisher` value of the parent query's candidate instance.

```
Query q = pm.newQuery (Magazine.class, "price == (select max(m.price) "
    + "from org.mag.Magazine m where m.publisher == publisher)");
List mags = (List) q.execute ();
```

The previous example used subqueries that were guaranteed to return exactly one result. What if the subquery might return no results, or multiple results? In these cases, you should treat the subquery as a collection. Use `contains` to test whether a value is included in the subquery results, and `isEmpty` to test whether the subquery has no results.

Example 9.14. Subquery Contains

Find all magazines whose title matches the title of an article.

```
Query q = pm.newQuery (Magazine.class,
    "(select a.title from org.mag.Article a).contains (title)");
```

Filtering on whether a subquery contains a value is the same as using the `IN` operator in SQL.

Example 9.15. Subquery Empty

Find all magazines whose title does not match the title of an article.

```
Query q = pm.newQuery (Magazine.class,
    "(select from org.mag.Article a where a.title == title).isEmpty ()");
```

Testing whether a subquery is empty is equivalent to SQL's `NOT EXISTS` assertion. To perform an `EXISTS` test instead, just negate the expression: `!(<subquery>).isEmpty ()`

9.6.4.1. Subquery Parameters, Variables, and Imports

Subqueries cannot declare their own parameters, variables, or imports. The outermost query must always specify all declarations for any subqueries it has, including nested subqueries. Subqueries can, however, introduce new implicit variables and parameters just by using them. For a refresher on JDOQL declarations, including implicit parameters and variables, see [Section 11.3, “Advanced Object Filtering” \[236\]](#) of the JDO Overview.

9.6.5. MethodQL

If JPQL / JDOQL and SQL queries do not match your needs, Kodo also allows you to name a Java method to use to load a set of objects. In a *MethodQL* query, the query string names a static method to invoke to determine the matching objects:

Important

This feature requires **Kodo Enterprise Edition**.

JPA:

```
import kodo.persistence.*;
...
// the method query language is 'kodo.MethodQL'.
// set the query string to the method to execute, including full class name; if
```

```
// the class is in the candidate class' package or in the query imports, you
// can omit the package; if the method is in the candidate class, you can omit
// the class name and just specify the method name
KodoEntityManager kem = KodoPersistence.cast (emf);
KodoQuery q = kem.createQuery ("kodo.MethodQL", "com.xyz.Finder.getByName");

// set the type of objects that the method returns
q.setResultClass (Person.class);

// parameters are passed the same way as in standard queries
q.setParameter ("firstName", "Fred").setParameter ("lastName", "Lucas");

// this executes your method to get the results
List results = q.getResultList ();
```

JDO:

```
// the method query language is 'kodo.MethodQL'.
// set the filter to the method to execute, including full class name; if
// the class is in the candidate class' package or in the query imports, you
// can omit the package; if the method is in the candidate class, you can omit
// the class name and just specify the method name
Query q = pm.newQuery ("kodo.MethodQL", "com.xyz.Finder.getByName");

// set the type of objects that the method returns
q.setClass (Person.class);

// parameters are declared and passed the same way as in standard queries
q.declareParameters ("String firstName, String lastName");

// this executes your method to get the results
List results = (List) q.execute ("Fred", "Lucas");
```

For datastore queries, the method must have the following signature:

```
public static ResultObjectProvider xxx(StoreContext ctx,
    ClassMetaData meta, boolean subclasses, Map params, FetchConfiguration fetch)
```

The returned result object provider should produce objects of the candidate class that match the method's search criteria. If the returned objects do not have all fields in the given fetch configuration loaded, Kodo will make additional trips to the datastore as necessary to fill in the data for the missing fields.

In-memory execution is slightly different, taking in one object at a time and returning a boolean on whether the object matches the query:

```
public static boolean xxx(StoreContext ctx, ClassMetaData meta,
    boolean subclasses, Object obj, Map params, FetchConfiguration fetch)
```

In both method versions, the given params map contains the names and values of all the parameters for the query.

The `StoredProcQueries` class and `StoredProcMain` driver program in [Section 1.3.5, “Custom Mappings” \[62\]](#) demonstrate how you might implement your own custom query method, and how to execute it through the JDO Query interface.

9.7. Generators

The JPA Overview's [Chapter 12, Mapping Metadata \[118\]](#) details using generators to automatically populate identity fields in

JPA.

The JDO Overview demonstrates how to declare sequences in [Chapter 15, Mapping Metadata \[260\]](#). It describes JDO's `javax.jdo.Sequence` interface and how to obtain named sequences in JDO in [Chapter 16, Sequence \[318\]](#).

Kodo represents all generators internally with the `kodo.kernel.Seq` interface. This interface supplies all the context you need to create your own custom generators, including the current persistence environment, the JDBC `DataSource`, and other essentials. The `kodo.jdbc.kernel.AbstractJDBCSeq` helps you create custom JDBC-based sequences. Kodo also supplies the following built-in Seqs:

- `table`: This is Kodo's default implementation. It is an alias for the `kodo.jdbc.kernel.TableJDBCSeq` class. The `TableJDBCSeq` uses a special single-row table to store a global sequence number. If the table does not already exist, it is created the first time you run the [mapping tool's](#) on a class that requires it. You can also use the class' `main` method or the `sequencetable` shell/bat script to manipulate the table; see the `TableJDBCSeq.main` method Javadoc for usage details.

This Seq has the following properties:

- `Table`: The name of the sequence number table to use. Defaults to `KODO_SEQUENCE_TABLE`.
- `PrimaryKeyColumn`: The name of the primary key column for the sequence table. Defaults to `ID`.
- `SequenceColumn`: The name of the column that will hold the current sequence value. Defaults to `SEQUENCE_VALUE`.
- `Allocate`: The number of values to allocate on each database trip. Defaults to 50, meaning the class will set aside the next 50 numbers each time it accesses the sequence table, which in turn means it only has to make a database trip to get new sequence numbers once every 50 sequence number requests.
- `class-table`: This is an alias for the `kodo.jdbc.kernel.ClassTableJDBCSeq`. This Seq is like the `TableJDBCSeq` above, but maintains a separate table row, and therefore a separate sequence number, for each base persistent class. It has all the properties of the `TableJDBCSeq`. Its table name defaults to `KODO_SEQUENCES_TABLE`. It also adds the following properties:
 - `IgnoreUnmapped`: Whether to ignore unmapped base classes, and instead use one row per least-derived mapped class. Defaults to `false`.
 - `UseAliases`: Whether to use each class' entity name as the primary key value of each row, rather than the full class name. Defaults to `false`.

As with the `TableJDBCSeq`, the `ClassTableJDBCSeq` creates its table automatically during mapping tool runs. However, you can manually manipulate the table through the class' `main` method, or through the `classsequencetable` shell/bat script. See the Javadoc for the `ClassTableJDBCSeq.main` method for usage details.

- `value-table`: This is an alias for the `kodo.jdbc.kernel.ValueTableJDBCSeq`. This Seq is like the `ClassTableJDBCSeq` above, but has an arbitrary number of rows for sequence values, rather than a fixed pattern of one row per class. Its table defaults to `KODO_SEQUENCES_TABLE`. It has all the properties of the `TableJDBCSeq`, plus:
 - `PrimaryKeyValue`: The primary key value used by this instance.

As with the `TableJDBCSeq`, the `ValueTableJDBCSeq` creates its table automatically during mapping tool runs. However, you can manually manipulate the table through the class' `main` method, or through the `valuesequencetable` shell/bat script. See the Javadoc for the `ValueTableJDBCSeq.main` method for usage details.

- `native`: This is an alias for the `kodo.jdbc.kernel.NativeJDBCSeq`. Many databases have a concept of "native sequences" - a built-in mechanism for obtaining incrementing numbers. For example, in Oracle, you can create a database sequence with a statement like `CREATE SEQUENCE MYSEQUENCE`. Sequence values can then be atomically obtained and incremented with the statement `SELECT MYSEQUENCE.NEXTVAL FROM DUAL`. Kodo provides support for this common mechanism of sequence generation with the `NativeJDBCSeq`, which accepts the following properties:
 - `Sequence`: The name of the database sequence. Defaults to `KODO_SEQUENCE`.

- `InitialValue`: The initial sequence value. Defaults to 1.
- `Increment`: The amount the sequence increments. Defaults to 1.
- `Allocate`: Some database can allocate values in-memory to service subsequent sequence requests faster.
- `time`: This is an alias for the `kodo.kernel.TimeSeededSeq`. This type uses an in-memory static counter, initialized to the current time in milliseconds and monotonically incremented for each value requested. It is only suitable for single-JVM environments.

You can use JPA `SequenceGenerators` to describe any built-in `Seqs` or your own `Seq` implementation. Set the `sequenceName` attribute to a plugin string describing your choice. See [Section 12.5, “Generators” \[126\]](#) in the JPA Overview for details on defining `SequenceGenerators`.

You can create named JDO sequences using any of the built-in `Seqs` or your own `Seq` implementation. Just set the sequence element's `factory-class` attribute to a plugin string describing your choice. See [Section 15.3, “Sequences” \[263\]](#) in the JDO Overview for details on defining named sequences in mapping metadata.

See [Section 2.5, “Plugin Configuration” \[395\]](#) for plugin string formatting.

Example 9.16. Named Seq Sequence

JPA:

```
@Entity
@Table(name="AUTO")
public class Author
{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="AuthorSeq")
    @SequenceGenerator(name="AuthorSeq" sequence="table(Table=AUTO_SEQ, Increment=100)")
    @Column(name="AID")
    private long id;
    ...
}
```

Note that if you want to use a plugin string without any arguments, you must still suffix the plugin type with `()` to differentiate it from a sequence name in the `SequenceGenerator.sequence` attribute:

```
@SequenceGenerator(name="AuthorSeq", sequence="table()")
```

JDO:

```
<?xml version="1.0"?>
<orm>
  <package name="org.mag.pub">
    <sequence name="AuthorSeq" factory-class="table(Table=AUTH_SEQ, Increment=100)"/>
    <class name="Author" table="AUTH">
      <datastore-identity sequence="AuthorSeq" column="AID"/>
      ...
    </class>
  </package>
</orm>
```

Kodo maintains a *system* sequence to generate datastore identity values for classes that do not declare a specific datastore identity strategy. You can configure the system sequence through the `kodo.Sequence` configuration property. This property accepts a plugin string describing a `Seq` instance.

Example 9.17. System Sequence Configuration

JPA XML format:

```
<property name="kodo.Sequence" value="table(Table=KODOSEQ, Increment=100)"/>
```

JDO properties format:

```
kodo.Sequence: table(Table=KODOSEQ, Increment=100)
```

In JPA, set your `GeneratedValue` annotation's `strategy` attribute to `AUTO` to use the configured system sequence. Or, because `AUTO` is the default strategy, use the annotation without attributes:

```
@GeneratedValue
private long id;
```

In JDO, set the `sequence` attribute of your `datastore-identity` or `field` element to `system` to use the system sequence:

```
<field name="id" primary-key="true" sequence="system"/>
```

9.7.1. Runtime Access

Kodo JPA allows you to access named generators at runtime through the `KodoEntityManager.getNamedGenerator` method:

```
public Generator getNamedGenerator (String name);
```

The returned `kodo.persistence.Generator` is a facade over an internal Kodo `Seq`.

The `KodoEntityManager` includes additional APIs to retrieve the identity generator of any class, or the generator of any field. With these APIs, you do not have to know the generator name. Additionally, they allow you to access the implicit generator used by default for datastore identity classes. See the **Javadoc** for the `KodoEntityManager.getIdentityGenerator` and `KodoEntityManager.getFieldGenerator` methods for API details.

The JDO Overview demonstrates how to obtain a sequence by name in **Chapter 16, Sequence [318]**. The `KodoJDOHelper` includes additional APIs to retrieve the identity sequence of any class, or the value sequence of any field. With these APIs, you do not have to know the sequence name. Additionally, they allow you to access the implicit sequence used for a datastore identity class with the default `native` strategy, or the backing sequence of a field with a value strategy of `uuid-hex` or `uuid-string`.

See the **Javadoc** for the `KodoJDOHelper.getIdentitySequence` and `KodoJDOHelper.getFieldSequence` methods for API details. Note that all methods return instances of `javax.jdo.Sequence` to coincide with JDO's standard APIs; the `Seq` interface is only used internally.

9.8. Transaction Events

The Kodo runtime supports broadcasting transaction-related events. By registering one or more `kodo.event.TransactionListeners`, you can receive notifications when transactions begin, flush, rollback, commit, and more. Where appropriate, event notifications include the set of persistence-capable objects participating in the transaction.

```
public void addTransactionListener (Object listener);
public void removeTransactionListener (Object listener);
```

These methods, available on any `KodoEntityManager` or `KodoPersistenceManager`, allow you to add and remove listeners.

For details on the transaction framework, see the `kodo.event` package **Javadoc**. Also see **Section 11.3, “Remote Event Notification Framework” [587]** for a description of Kodo's remote event support.

9.9. Non-Relational Stores

It is possible to adapt Kodo to access a non-relational datastore by creating an implementation of the `kodo.kernel.StoreManager` interface. Kodo provides an abstract `StoreManager` implementation to facilitate this process. See the `kodo.abstractstore` package **Javadoc** for details. Additionally, see **Section 1.3.12, “XML Store Manager” [624]** for an example of how to extend this abstract store manager.

Important

This feature requires **Kodo Enterprise Edition**.

Chapter 10. Caching

Kodo utilizes several configurable caches to maximize performance. This chapter explores Kodo's data cache, query cache, and query compilation cache.

10.1. Data Cache

The Kodo data cache is an optional cache of persistent object data that operates at the `EntityManagerFactory / PersistenceManagerFactory` level. This cache is designed to significantly increase performance while remaining in full compliance with the JPA and JDO persistence standards. This means that turning on the caching option can transparently increase the performance of your application, with no changes to your code.

Important

This feature requires **Kodo Enterprise Edition** or **Kodo Standard Edition** with the **Performance Pack**.

Kodo's data cache is not related to the `EntityManager` or `PersistenceManager` caches dictated by the JPA and JDO persistence specifications. These specifications mandate behavior for the `EntityManager` and `PersistenceManager` caches aimed at guaranteeing transaction isolation when operating on persistent objects.

Kodo's data cache is designed to provide significant performance increases over cacheless operation, while guaranteeing that behavior will be identical in both cache-enabled and cacheless operation.

There are five ways to access data via the Kodo APIs: standard relation traversal, large result set relation traversal, queries, looking up an object by id, and iteration over an `Extent`. Kodo's cache plugin accelerates three of these mechanisms. It does not provide any caching of large result set relations or `Extent` iterators. If you find yourself in need of higher-performance `Extent` iteration, see **Example 10.20, “Query Replaces Extent”** [575]

Table 10.1. Data access methods

Access method	Uses cache
Standard relation traversal	Yes
Large result set relation traversal	No
Query	Yes
Lookups by object id	Yes
Iteration over an <code>Extent</code>	No

When enabled, the cache is checked before making a trip to the datastore. Data is stored in the cache when objects are committed and when persistent objects are loaded from the datastore.

Kodo's data cache can in both single-JVM and multi-JVM environments. Multi-JVM caching is achieved through the use of the distributed event notification framework described in **Section 11.3, “Remote Event Notification Framework”** [587] or through one of Kodo's integrations with third-party distributed caches (see **Section 10.1.5, “Third-Party Integrations”** [572])

The single JVM mode of operation maintains and shares a data cache across all `EntityManager` or `PersistenceManager` instances obtained from a particular `EntityManagerFactory` or `PersistenceManagerFactory`. This is not appropriate for use in a distributed environment, as caches in different JVMs or created from different factory objects will not be synchronized.

Note

If you mix and match `EntityManagers` and `PersistenceManagers` in the same application, they will use the same data cache so long as you obtain every manager from the same factory, or as long as your `EntityManagerFactory` and `PersistenceManagerFactory` share the same underlying `BrokerFactory`.

10.1.1. Data Cache Configuration

To enable the basic single-factory cache set the `kodo.DataCache` property to `true`, and set the `kodo.RemoteCommitProvider` property to `sjvm`:

Example 10.1. Single-JVM Data Cache

JPA XML format:

```
<property name="kodo.DataCache" value="true"/>
<property name="kodo.RemoteCommitProvider" value="sjvm"/>
```

JDO properties format:

```
kodo.DataCache: true
kodo.RemoteCommitProvider: sjvm
```

To configure the data cache to remain up-to-date in a distributed environment, set the `kodo.RemoteCommitProvider` property appropriately, or integrate Kodo with a third-party caching solution. Remote commit providers are described in [Section 11.3, “Remote Event Notification Framework” \[587\]](#) and [Section 10.1.5, “Third-Party Integrations” \[572\]](#) enumerates supported third-party caching solutions.

Kodo's default implementation maintains a least-recently-used map of object ids to cache data. By default, 1000 elements are kept in cache. This can be adjusted by setting the `CacheSize` property in your plugin string - see below for an example. Objects that are pinned into the cache are not counted when determining if the cache size exceeds the maximum.

Expired objects are moved to a soft reference map, so they may stick around for a little while longer. You can control the number of soft references Kodo keeps with the `SoftReferenceSize` property. Soft references are unlimited by default. Set to 0 to disable soft references completely.

Example 10.2. Data Cache Size

JPA XML format:

```
<property name="kodo.DataCache" value="true(CacheSize=5000, SoftReferenceSize=0)"/>
```

JDO properties format:

```
kodo.DataCache: true(CacheSize=5000, SoftReferenceSize=0)
```

You can specify a cache timeout value for a class by setting the timeout **metadata extension** to the amount of time in milliseconds a class's data is valid. Use a value of -1 for no expiration. This is the default value.

Example 10.3. Data Cache Timeout

Timeout Employee objects after 10 seconds.

JPA:

```
@Entity
@DataCache(timeout=10000)
public class Employee
{
    ...
}
```

JDO:

```
<class name="Employee">
    <extension vendor-name="kodo" key="data-cache-timeout" value="10000"/>
    ...
</class>
```

See the **kodo.persistence.DataCache** Javadoc for more information on the DataCache annotation.

A cache can specify that it should be cleared at certain times rather than using data timeouts. The **EvictionSchedule** property of Kodo's cache implementation accepts a **cron** style eviction schedule. The format of this property is a whitespace-separated list of five tokens, where the ***** symbol (asterisk), indicates match all. The tokens are, in order:

- Minute
- Hour of Day
- Day of Month
- Month
- Day of Week

For example, the following **kodo.DataCache** setting schedules the default cache to evict values from the cache at 15 and 45 minutes past 3 PM on Sunday.

```
true(EvictionSchedule='15,45 15 * * 1')
```

It is also possible for different persistence-capable classes to use different caches. This is achieved by specifying a cache name in a **metadata extension**.

Example 10.4. Named Data Cache Specification

JPA:

```
import kodo.persistence.*;

@Entity
@DataCache(name="small-cache", timeout=10000)
public class Employee
{
    ...
}
```

JDO:

```
<class name="Employee">
  <extension vendor-name="kodo" key="data-cache" value="small-cache"/>
  <extension vendor-name="kodo" key="data-cache-timeout" value="10000"/>
  ...
</class>
```

See the `kodo.persistence.DataCache` Javadoc for more information on the `DataCache` annotation.

The metadata above will cause instances of the `Employee` class to be stored in a cache named `small-cache`. This `small-cache` cache can be explicitly configured in the `kodo.DataCache` plugin string, or can be implicitly defined, in which case it will take on the same default configuration properties as the default cache identified in the `kodo.DataCache` property.

Example 10.5. Named Data Cache Configuration

JPA XML format:

```
<property name="kodo.DataCache" value="true, true(Name=small-cache, CacheSize=100)"/>
```

JDO properties format:

```
kodo.DataCache: true, true(Name=small-cache, CacheSize=100)
```

10.1.2. Data Cache Usage

The `kodo.datacache` package defines Kodo's data caching framework. While you may use this framework directly (see its **Javadoc** for details), its APIs are meant primarily for service providers. In fact, **Section 10.1.6, "Cache Extension" [574]** below has tips on how to use this package to extend Kodo's caching service yourself.

Rather than use the low-level `kodo.datacache` package APIs, JPA users should typically access the data cache through Kodo's high-level `kodo.persistence.StoreCache` facade. This facade has methods to pin and unpin records, evict data from the cache, and more.

```
public StoreCache getStoreCache ();
public StoreCache getStoreCache (String name);
```

You obtain the `StoreCache` through the `KodoEntityManagerFactory.getStoreCache` methods. When you have multiple data caches configured as in the `small-cache` example above, the `StoreCache` can act as a unified facade over all your caches. For every `oid` parameter to the `StoreCache` methods, it determines the correct data cache for that `oid`'s corresponding persistent class, and dynamically delegates to that cache.

If you know that you want to access a certain data cache and no others, the `KodoEntityManagerFactory.getStoreCache(String name)` method returns a `StoreCache` interface to a particular named data cache.

Example 10.6. Accessing the StoreCache

```
import kodo.persistence.*;
...
KodoEntityManagerFactory kemf = KodoPersistence.cast (emf);
StoreCache cache = kemf.getStoreCache ();
...
StoreCache smallCache = kemf.getStoreCache ("small-cache");
...
```

```
public void evict (Class cls, Object oid);
public void evictAll ();
public void evictAll (Class cls, Object... oids);
public void evictAll (Class cls, Collection oids);
```

The `evict` methods tell the cache to release data. Each method takes an entity class and one or more identity values, and releases the cached data for the corresponding persistent instances. The `evictAll` method with no arguments clears the cache. Eviction is useful when the datastore is changed by a separate process outside Kodo's control. In this scenario, you typically have to manually evict the data from the datastore cache; otherwise the Kodo runtime, oblivious to the changes, will maintain its stale copy.

```
public void pin (Class cls, Object oid);
public void pinAll (Class cls, Object... oids);
public void pinAll (Class cls, Collection oids);
public void unpin (Class cls, Object oid);
public void unpinAll (Class cls, Object... oids);
public void unpinAll (Class cls, Collection oids);
```

Most caches are of limited size. Pinning an identity to the cache ensures that the cache will not kick the data for the corresponding instance out of the cache, unless you manually evict it. Note that even after manual eviction, the data will get pinned again the next time it is fetched from the store. You can only remove a pin and make the data once again available for normal cache overflow eviction through the `unpin` methods. Use pinning when you want a guarantee that a certain object will always be

available from cache, rather than requiring a datastore trip.

Example 10.7. StoreCache Usage

```
import kodo.persistence.*;
...
KodoEntityManagerFactory kemf = KodoPersistence.cast (emf);
StoreCache cache = kemf.getStoreCache ();
cache.pin (Magazine.class, popularMag.getId ());
cache.evict (Magazine.class, changedMag.getId ());
```

See the `StoreCache` **Javadoc** for information on additional functionality it provides. Also, **Chapter 9, Runtime Extensions [538]** discusses Kodo's other extensions to the standard set of JPA runtime interfaces.

JDO users can access the data cache through the JDO-standard `javax.jdo.datastore.DataStoreCache` facade. The JDO Overview describes this interface in **Chapter 13, DataStoreCache [257]**. Kodo extends the standard JDO cache in two ways:

1. The `kodo.jdo.KodoDataStoreCache` interface extends JDO's `DataStoreCache` with many useful methods.
2. When you have multiple data caches configured as in the `small-cache` example above, Kodo's `DataStoreCache` implementation acts as a facade over all your data caches. For every oid parameter passed to the `DataStoreCache` methods, it determines the correct data cache for that oid's corresponding persistent class, and dynamically delegates to that cache.

If you know that you want to access a certain data cache and no others, the `KodoPersistenceManagerFactory.getDataStoreCache(String name)` method returns a JDO `DataStoreCache` interface to a particular named data cache.

Example 10.8. Accessing a Named Cache

```
import kodo.jdo.*;
...
KodoPersistenceManagerFactory kpmf = KodoJDOHelper.cast (pmf);
DataStoreCache cache = kpmf.getDataStoreCache ("small-cache");
cache.evict (JDOHelper.getObjectId (changedObj));
cache.pin (JDOHelper.getObjectId (popularObj));
```

You can read more about Kodo extensions to JDO APIs in **Chapter 9, Runtime Extensions [538]**.

The examples above include calls to `evict` to manually remove data from the data cache. Rather than evicting objects from the data cache directly, you can also configure Kodo to automatically evict objects from the data cache when you use the `KodoEntityManager` or `PersistenceManager`'s eviction APIs.

Example 10.9. Automatic Data Cache Eviction

JPA XML format:

```
<property name="kodo.BrokerImpl" value="EvictFromDataCache=true"/>
```

JDO properties format:

```
kodo.BrokerImpl: EvictFromDataCache=true
```

JPA:

```
import kodo.persistence.*;
...
KodoEntityManager kem = KodoPersistence.cast (em);
kem.evict (changedMag); // will evict from data cache also
```

JDO:

```
PersistenceManager pm = ...;
pm.evict (changedMag); // will evict from data cache also
```

10.1.3. Query Cache

In addition to the data cache, the `kodo.datacache` package defines service provider interfaces for a query cache. The query cache is enabled by default when the data cache is enabled. The query cache stores the object ids returned by query executions. When you run a query, Kodo assembles a key based on the query properties and the parameters used at execution time, and checks for a cached query result. If one is found, the object ids in the cached result are looked up, and the resultant persistence-capable objects are returned. Otherwise, the query is executed against the database, and the object ids loaded by the query are put into the cache. The object id list is not cached until the list returned at query execution time is fully traversed.

Kodo JPA exposes a high-level interface to the query cache through the `kodo.persistence.QueryResultCache` class. You can access this class through the `KodoEntityManagerFactory`.

Kodo JDO exposes a high-level interface to the query cache through the `kodo.jdo.QueryResultCache` class. You can access this class through the `KodoPersistenceManagerFactory`.

Example 10.10. Accessing the QueryResultCache

JPA:

```
import kodo.persistence.*;
...
KodoEntityManagerFactory kemf = KodoPersistence.cast (emf);
```

```
QueryResultCache qcache = kemf.getQueryResultCache ();
```

JDO:

```
import kodo.jdo.*;
...
KodoPersistenceManagerFactory kpmf = KodoJDOHelper.cast (pmf);
QueryResultCache qcache = kpmf.getQueryResultCache ();
```

The default query cache implementation caches 100 query executions in a least-recently-used cache. This can be changed by setting the cache size in the `CacheSize` plugin property. Like the data cache, the query cache also has a backing soft reference map. The `SoftReferenceSize` property controls the size of this map. It is disabled by default.

Example 10.11. Query Cache Size

JPA XML format:

```
<property name="kodo.QueryCache" value="CacheSize=1000, SoftReferenceSize=100"/>
```

JDO properties format:

```
kodo.QueryCache: CacheSize=1000, SoftReferenceSize=100
```

To disable the query cache completely, set the `kodo.QueryCache` property to `false`:

Example 10.12. Disabling the Query Cache

JPA XML format:

```
<property name="kodo.QueryCache" value="false"/>
```

JDO properties format:

```
kodo.QueryCache: false
```

There are certain situations in which the query cache is bypassed:

- Caching is not used for in-memory queries (queries in which the candidates are a collection instead of a class or `Extent`).
- Caching is not used in transactions that have `IgnoreChanges` set to `false` and in which modifications to classes in the query's access path have occurred. If none of the classes in the access path have been touched, then cached results are still valid and are used.
- Caching is not used in pessimistic transactions, since Kodo must go to the database to lock the appropriate rows.
- Caching is not used when the the data cache does not have any cached data for an id in a query result.
- Queries that use persistence-capable objects as parameters are only cached if the parameter is directly compared to field, as in:

JPQL:

```
select e from Employee e where e.company.address = :addr
```

JDOQL:

```
select from com.xyz.Employee where company.address == :addr
```

If you extract field values from the parameter in your query string, or if the parameter is used in collection element comparisons, the query is not cached.

- Queries that result in projections of custom field types or `BigDecimal` or `BigInteger` fields are not cached.

Cache results are removed from the cache when instances of classes in a cached query's access path are touched. That is, if a query accesses data in class A, and instances of class A are modified, deleted, or inserted, then the cached query result is dropped from the cache.

It is possible to tell the query cache that a class has been altered. This is only necessary when the changes occur via direct modification of the database outside of Kodo's control. You can also evict individual queries, or clear the entire cache.

JPA:

```
public void evict (Query q);  
public void evictAll (Class cls);  
public void evictAll ();
```

JDO:

```
public void evict (Query q);  
public void evict (Query q, Object[] params);  
public void evict (Query q, Map params);  
public void evictAll (Class cls);  
public void evictAll ();
```

For JPA queries with parameters, set the desired parameter values into the `Query` instance before calling the above methods.

For JDO queries with parameters, pass the parameter values of the query in the `params` array or map.

Example 10.13. Evicting Queries

JPA:

```
import kodo.persistence.*;
...
KodoEntityManagerFactory kemf = KodoPersistence.cast (emf);
QueryResultCache qcache = kemf.getQueryResultCache ();

// evict all queries that can be affected by changes to Magazines
qcache.evictAll (Magazine.class);

// evict an individual query with parameters
EntityManager em = emf.createEntityManager ();
Query q = em.createQuery (...).
    setParameter (0, paramVal0).
    setParameter (1, paramVal1);
qcache.evict (q);
```

JDO:

```
import kodo.jdo.*;
...
KodoPersistenceManagerFactory kpmf = KodoJDOHelper.cast (pmf);
QueryResultCache qcache = kpmf.getQueryResultCache ();

// evict all queries that can be affected by changes to Magazines
qcache.evictAll (Magazine.class);

// evict an individual query with parameters
PersistenceManager pm = pmf.getPersistenceManager ();
Query q = pm.newQuery (...);
qcache.evictAll (q, new Object[] { paramVal0, paramVal1 });
```

When using one of Kodo's distributed cache implementations, it is necessary to perform this in every JVM - the change notification is not propagated automatically. When using a coherent cache implementation such as Kodo's Tangosol cache implementation, it is not necessary to do this in every JVM (although it won't hurt to do so), as the cache results are stored directly in the coherent cache.

Queries can also be pinned and unpinned through the `QueryResultCache`. The semantics of these operations are the same as pinning and unpinning data from the data cache.

JPA:

```
public void pin (Query q);
public void unpin (Query q);
```

JDO:

```

public void pin (Query q);
public void pin (Query q, Object[] params);
public void pin (Query q, Map params);
public void unpin (Query q);
public void unpin (Query q, Object[] params);
public void unpin (Query q, Map params);

```

For JPA queries with parameters, set the desired parameter values into the **Query** instance before calling the above methods.

For JDO queries with parameters, pass the parameter values of the query in the `params` array or map.

The following example shows these APIs in action.

Example 10.14. Pinning, and Unpinning Query Results

JPA:

```

import kodo.persistence.*;
...
KodoEntityManagerFactory kemf = KodoPersistence.cast (emf);
QueryResultCache qcache = kemf.getQueryResultCache ();
EntityManager em = emf.createEntityManager ();

Query pinQuery = em.createQuery (...).
    setParameter (0, paramVal0).
    setParameter (1, paramVal1);
qcache.pin (pinQuery);
Query unpinQuery = em.createQuery (...).
    setParameter (0, paramVal0).
    setParameter (1, paramVal1);
qcache.unpin (unpinQuery);

```

JDO:

```

import kodo.jdo.*;
...
KodoPersistenceManagerFactory kpmf = KodoJDOHelper.cast (pmf);
QueryResultCache qcache = kpmf.getQueryResultCache ();
PersistenceManager pm = pmf.getPersistenceManager ();

Query pinQuery = pm.newQuery (...);
qcache.pin (pinQuery, new Object[] { paramVal0, paramVal1 });
Query unpinQuery = pm.newQuery (...);
qcache.unpin (unpinQuery, new Object[] { paramVal0, paramVal1 });

```

Pinning data into the cache instructs the cache to not expire the pinned results when cache flushing occurs. However, pinned results will be removed from the cache if an event occurs that invalidates the results.

You can disable caching on a per-EntityManager, per-PersistenceManager or per-Query basis:

Example 10.15. Disabling and Enabling Query Caching

JPA:

```
import kodo.persistence.*;
...
// temporarily disable query caching for all queries created from em
KodoEntityManager kem = KodoPersistence.cast (em);
kem.getFetchPlan ().setQueryResultCache (false);

// re-enable caching for a particular query
KodoQuery kq = kem.createQuery (...);
kq.getFetchPlan ().setQueryResultCache (true);
```

JDO:

```
import kodo.jdo.*;
...
// temporarily disable query caching for all queries created from pm
PersistenceManager pm = ...;
KodoFetchPlan fetch = (KodoFetchPlan) pm.getFetchPlan ();
fetch.setQueryResultCache (false);

// re-enable caching for a particular query
Query q = pm.newQuery (...);
KodoFetchPlan fetch = KodoJDOHelper.cast (pm.getFetchPlan ());
fetch.setQueryResultCache (true);
```

10.1.4. The Concurrent Data Cache

The *concurrent* is a new data cache plugin for Kodo. It has not been as thoroughly tested as Kodo's standard data cache. The concurrent cache can, however, offer considerably greater performance and scalability characteristics compared to the standard cache when the following conditions are met:

1. The cache is big enough to hold all of your commonly-used data.
2. Your application is heavily concurrent.

Additionally, this cache does not fully index its contents by class, but rather just keeps track of which classes are in the cache. It services requests to drop given classes by checking to see if any instances of that class might be in the cache, and then clearing the entire cache. This inefficiency can easily be worked around with careful cache partitioning.

To use the concurrent data cache instead of the standard data cache, set your `kodo.DataCache` property to `concurrent` instead of `true`. Or, you can mix concurrent and standard caches in a set of named caches.

The concurrent cache has the following configuration parameters:

1. `CacheSize`: The maximum number of objects that this cache will hold. The default is 1000 objects. This setting differs from the setting for the default data cache in that instances that are pinned into cache do count towards this limit.
2. `Name`: The unique name of the cache.
3. `EvictionSchedule`: A cron-style eviction schedule that this cache should follow.

4. `LRUThresholdRatio`: The ratio above which this implementation should start applying its least-recently-used algorithm to prepare for eviction should the need arise. Default is 80%. Values should be integers between 1 and 100.

Example 10.16. Concurrent Data Cache Configuration

The following configuration uses the standard Kodo cache as the primary cache and a named concurrent cache as a secondary cache. Various classes can be configured to use this secondary cache in their metadata.

JPA XML format:

```
<property name="kodo.DataCache" value="true, concurrent(Name=secondary)"/>
```

JDO properties format:

```
kodo.DataCache: true, concurrent(Name=secondary)
```

10.1.5. Third-Party Integrations

Kodo includes built-in integrations with Tangosol Coherence and GemStone GemFire caching products.

10.1.5.1. Tangosol Integration

The Kodo data cache can integrate with Tangosol's Coherence caching system. To use Tangosol integration, set the `kodo.DataCache` configuration property to `tangosol`, with the appropriate plugin properties for your Tangosol setup. For example:

Example 10.17. Tangosol Cache Configuration

JPA XML format:

```
<property name="kodo.DataCache" value="tangosol(TangosolCacheName=kodo)"/>
```

JDO properties format:

```
kodo.DataCache: tangosol(TangosolCacheName=kodo)
```

The Tangosol cache understands the following properties:

- `TangosolCacheName`: The name of the Tangosol Coherence cache to use. Defaults to `kodo`.
- `TangosolCacheType`: The type of Tangosol Coherence cache to use (optional). Valid values are `named`, `distributed`, or `replicated`. Defaults to `named`, which means that the cache is looked up via the `com.tangosol.net.CacheFactory.getCache(String)` method. This method looks up the cache by name as defined in the Coherence configuration.

Note

As of this writing, it is not possible to use a Tangosol Coherence 1.2.2 distributed cache type with Apple's OS X 1.3.1 JVM. Use their replicated cache instead.

- `ClearOnClose`: Whether the Tangosol named cache should be completely cleared when the `EntityManagerFactory` or `PersistenceManagerFactory` is closed. Defaults to `false`.

The Kodo query cache can also integrate with Tangosol's Coherence caching system. To use Tangosol query cache integration, set the `kodo.QueryCache` configuration property to `tangosol`, with the appropriate plugin properties for your Tangosol setup. For example:

Example 10.18. Tangosol Query Cache Configuration

JPA XML format:

```
<property name="kodo.QueryCache" value="tangosol(TangosolCacheName=kodo-query)"/>
```

JDO properties format:

```
kodo.QueryCache: tangosol(TangosolCacheName=kodo-query)
```

The Tangosol query cache understands the same properties as the data cache, with a default Tangosol cache name of `kodo-query`.

10.1.5.2. GemStone GemFire Integration

The Kodo data cache can integrate with GemStone's GemFire v3.5.1 caching system. To use GemFire in Kodo you will need to change your `gemfire.properties` to have the property `enable-shared-memory=true`. You will also need to add both Kodo and GemFire to your classpath and then start a GemFire server.

```
prompt> gemfire start
```

By default, the GemFire data cache will use a GemFire region of `root/kodo-data-cache` and the GemFire query cache will use a region of `root/kodo-query-cache`. This can be changed by setting the optional property `GemFireCacheName`.

Example 10.19. GemFire Cache Configuration

JPA persistence.xml:

```
<property name="kodo.DataCache"
  value="gemfire(GemFireCacheName=/root/my-kodo-data-cache)"/>
<property name="kodo.QueryCache"
  value="gemfire(GemFireCacheName=/root/my-kodo-query-cache)"/>
```

JDO properties file:

```
kodo.DataCache: gemfire(GemFireCacheName=/root/My-kodo-data-cache)
kodo.QueryCache: gemfire(GemFireCacheName=/root/My-kodo-query-cache)
```

GemFire cache.xml:

```
...
<shared-root-region name="root">
  <region-attributes>
    ...
  </region-attributes>
  <region name="My-kodo-data-cache">
    <region-attributes>
      </region-attributes>
    </region>
  <region name="My-kodo-query-cache">
    <region-attributes>
      </region-attributes>
    </region>
  </shared-root-region>
...
```

If you set GemFire for both `kodo.DataCache` and `kodo.QueryCache` you aren't required to specify a `kodo.RemoteCommitProvider` unless you are registering your own `RemoteCommitListeners`.

Some notes regarding using GemFire with Kodo:

- Custom field types mapped with externalizers or custom mappings must be serializable.
- The `kodo.DynamicDataStructs` option is not supported.

10.1.6. Cache Extension

The provided data cache classes can be easily extended to add additional functionality. If you are adding new behavior, you should extend `kodo.datacache.DataCacheImpl`. To use your own storage mechanism, extend `kodo.datacache.AbstractDataCache`, or implement `kodo.datacache.DataCache` directly. If you want to implement a distributed cache that uses an unsupported method for communications, create an implementation of `kodo.event.RemoteCommitProvider`. This process is described in greater detail in [Section 11.3.2, “Customization”](#) [?].

The query cache is just as easy to extend. Add functionality by extending the default `kodo.datacache.QueryCacheImpl`. Implement your own storage mechanism for query results by extending `kodo.datacache.AbstractQueryCache` or implementing the `kodo.datacache.QueryCache` interface directly.

10.1.7. Important Notes

- The default cache implementations *do not* automatically refresh objects in other `EntityManager`s and `PersistenceManager`s when the cache is updated or invalidated. This behavior would not be compliant with the JPA and JDO specifications.
- Invoking `KodoEntityManager.evict` and `PersistenceManager.evict` *does not* result in the corresponding data being dropped from the data cache, unless you have set the proper configuration options as explained above (see [Example 10.9, “Automatic Data Cache Eviction” \[565\]](#)). Other methods related to the `EntityManager` and `PersistenceManager` caches also do not effect the data cache.

The data cache assumes that it is up-to-date with respect to the datastore, so it is effectively an in-memory extension of the database. To manipulate the data cache, you should generally use the data cache facades presented in this chapter.

- You must specify a `kodo.event.RemoteCommitProvider` (via the `kodo.RemoteCommitProvider` property) in order to use the data cache, even when using the cache in a single-JVM mode. When using it in a single-JVM context, set this property to `sjvm`.

10.1.8. Known Issues and Limitations

- When using datastore (pessimistic) transactions in concert with the distributed caching implementations, it is possible to read stale data when reading data outside a transaction.

For example, if you have two JVMs (JVM A and JVM B) both communicating with each other, and JVM A obtains a data store lock on a particular object's underlying data, it is possible for JVM B to load the data from the cache without going to the datastore, and therefore load data that should be locked. This will only happen if JVM B attempts to read data that is already in its cache during the period between when JVM A locked the data and JVM B received and processed the invalidation notification.

This problem is impossible to solve without putting together a two-phase commit system for cache notifications, which would add significant overhead to the caching implementation. As a result, we recommend that people use optimistic locking when using data caching. If you do not, then understand that some of your non-transactional data may not be consistent with the datastore.

Note that when loading objects in a transaction, the appropriate datastore transactions will be obtained. So, transactional code will maintain its integrity.

- `Extents` are not cached. So, if you plan on iterating over a list of all the objects in an `Extent` on a regular basis, you will only benefit from caching if you do so with a `Query` instead:

Example 10.20. Query Replaces Extent

JPA:

```
import kodo.persistence.*;
...
KodoEntityManager kem = KodoPersistence.cast (em);
Extent extent = kem.getExtent (Magazine.class, false);

// This iterator does not benefit from caching...
```

```

Iterator uncachedIterator = extent.iterator ();

// ... but this one does.
KodoQuery extentQuery = kem.createQuery (...);
extentQuery.setSubclasses (false);
Iterator cachedIterator = extentQuery.getResultList ().iterator ();

```

JDO:

```

Extent extent = pm.getExtent (Magazine.class, false);

// This iterator does not benefit from caching...
Iterator uncachedIterator = extent.iterator ();

// ... but this one does.
Query extentQuery = pm.newQuery (extent);
Iterator cachedIterator = ((List) extentQuery.execute ().iterator ());

```

10.2. Query Compilation Cache

The query compilation cache is a `Map` used to cache parsed query strings. As a result, most queries are only parsed once in Kodo, and cached thereafter. You can control the compilation cache through the `kodo.QueryCompilationCache` configuration property. This property accepts a plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) describing the `Map` used to associate query strings and their parsed form. This property accepts the following aliases:

Table 10.2. Pre-defined aliases

Alias	Value
true	<code>kodo.util.CacheMap</code>
all	<code>java.util.HashMap</code>
false	<i>none</i>

Chapter 11. Remote and Offline Operation

The standard JPA and JDO runtime environments are *local* and *online*. They are *local* in that components such as `EntityManager`s, `PersistenceManager`s, and queries connect directly to the datastore and execute their actions in the same JVM as the code using them. They are *online* in that changes to managed objects take place in the context of an active `EntityManager` or `PersistenceManager`. These two properties, combined with the fact that managers cannot be serialized for storage or network transfer, make the standard JPA and JDO runtimes difficult to incorporate into some enterprise and client/server program designs.

Kodo extends the standard runtime to add *remote* and *offline* capabilities in the form of enhanced **Detach and Attach APIs**, **Remote Managers**, and **Remote Commit Events**. The following sections explain these capabilities in detail.

11.1. Detach and Attach

The JPA Overview describes JPA's standard detach and attach APIs in [Chapter 8, *EntityManager* \[70\]](#). The JDO Overview does the same for JDO in [Section 8.7, “Detach and Attach Functionality” \[219\]](#). This section enumerates Kodo's enhancements to the standard behavior.

11.1.1. Detach Behavior

In JPA, objects detach automatically when they are serialized or when a **persistence context** ends. The specification does not define any way to explicitly detach objects. The extended `KodoEntityManager`, however, allows you to explicitly detach objects at any time.

```
public Object detach (Object pc):
public Object[] detachAll (Object... pcs):
public Collection detachAll (Collection pcs):
```

Each detach method returns detached copies of the given instances. The copy mechanism is similar to serialization, except that only certain fields are traversed. We will see how to control which fields are detached in a later section.

When detaching an instance that has been modified in the current transaction (and thus made dirty), the current transaction is flushed. This means that when subsequently re-attaching the detached instances, Kodo assumes that the transaction from which they were originally detached was committed; if it has been rolled back, then the re-attachment process will throw an optimistic concurrency exception.

You can stop Kodo from assuming the transaction will commit by invoking `setRollbackOnly` prior to detaching your objects. Setting the `RollbackOnly` flag prevents Kodo from flushing when detaching dirty objects; instead Kodo just runs its pre-flush actions (see the `KodoEntityManager.preFlush` or `KodoPersistenceManager.preFlush` Javadoc for details).

This allows you to use the same instances in multiple attach/modify/detach/rollback cycles. Alternatively, you might also prevent a flush by making your modifications outside of a transaction (with `NontransactionalWrite` enabled) before detaching.

11.1.2. Attach Behavior

When attaching, Kodo uses several strategies to determine the optimal way to merge changes made to the detached instance. As you will see, these strategies can even be used to attach changes made to a transient instance which was never detached in the first place.

- If the instance was detached and **detached state** is enabled, Kodo will use the detached state to determine the object's version

and primary key values. In addition, this state will tell Kodo which fields were loaded at the time of detach, and in turn where to expect changes. Loaded detached fields with null values will set the attached instance's corresponding fields to null.

- If the instance has an application visible version field, Kodo will consider the object detached if the version field has a non-default value, and new otherwise.

When attaching null fields in these cases, Kodo cannot distinguish between a field that was unloaded and one that was intentionally set to null. In this case, Kodo will use the current **detach state** setting to determine how to handle null fields: fields that would have been included in the detached state are treated as loaded, and will in turn set the corresponding attached field to null.

- If neither of the above cases apply, Kodo will check to see if an instance with the same primary key values exists in the database. If so, the object is considered detached. Otherwise, it is considered new.

These strategies will be assigned on a per-instance basis, such that during the attachment of an object graph more than one of the above strategies may be used.

If you attempt to attach a versioned instance whose representation has changed in the datastore since detachment, Kodo will throw an optimistic concurrency exception upon commit or flush, just as if a normal optimistic conflict was detected. When attaching an instance whose database record has been deleted since detaching, or when attaching a detached instance into a manager that has a stale version of the object, Kodo will throw an optimistic concurrency exception from the attach method. In these cases, Kodo sets the `RollbackOnly` flag on the transaction.

11.1.3. Defining the Detached Object Graph

When detached objects lose their association with the Kodo runtime, they also lose the ability to load additional state from the datastore. It is important, therefore, to populate objects with all the persistent state you will need before detaching them. While you are free to do this manually, Kodo includes facilities for automatically populating objects when they detach. The `kodo.DetachState` configuration property determines which fields and relations are detached by default. All settings are recursive. They are:

1. `loaded`: Detach all fields and relations that are already loaded, but don't include unloaded fields in the detached graph. This is the default.
2. `fgs`: Detach all fields and relations in the default fetch group, and any other fetch groups that you have added to the current **fetch configuration**. For more information on custom fetch groups, see [Section 5.6, “Fetch Groups” \[469\]](#).
3. `all`: Detach all fields and relations. Be very careful when using this mode; if you have a highly-connected domain model, you could end up bringing every object in the database into memory!

Any field that is not included in the set determined by the detach mode is set to its Java default value in the detached instance.

The `kodo.DetachState` option is actually a plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)) that allows you to also configure the following options related to detached state:

- `DetachedStateField`: As described in [Section 11.1.2, “Attach Behavior” \[577\]](#) above, Kodo can take advantage of a *detached state field* to make the attach process more efficient. This field is added by the enhancer and is not visible to your application. Set this property to one of the following values:
 - `transient`: Use a transient detached state field. This gives the benefits of a detached state field to local objects that are never serialized, but retains serialization compatibility for client tiers without access to the enhanced versions of your classes. This is the default.
 - `true`: Use a non-transient detached state field so that objects crossing serialization barriers can still be attached efficiently. This requires, however, that your client tier have the enhanced versions of your classes and the Kodo libraries.

- `false`: Do not use a detached state field.

You can override the setting of this property or declare your own detached state field on individual classes using Kodo's metadata extensions. See [Section 11.1.3.1, “Detached State Field” \[580\]](#) below.

- `DetachedStateManager`: Whether to use a detached state manager. A detached state manager makes attachment much more efficient. Like a detached state field, however, it breaks serialization compatibility with the unenhanced class if it isn't transient.

This setting piggybacks on the `DetachedStateField` setting above. If your detached state field is transient, the detached state manager will also be transient. If the detached state field is disabled, the detached state manager will also be disabled. This is typically what you'll want. By setting `DetachedStateField` to true (or transient) and setting this property to false, however, you can use a detached state field **without** using a detached state manager. This may be useful for debugging or for legacy Kodo users who find differences between Kodo's behavior with a detached state manager and Kodo's older behavior without one.

- `AccessUnloaded`: Whether to allow access to unloaded fields of detached objects. Defaults to true. Set to false to throw an exception whenever an unloaded field is accessed. This option is only available when you use detached state managers, as determined by the settings above.

Example 11.1. Configuring Detached State

JPA XML format:

```
<property name="kodo.DetachState" value="fgs(DetachedStateField=true)"/>
```

JDO properties format:

```
kodo.DetachState: fgs(DetachStateFielded=true)
```

You can also alter the set of fields that will be included in the detached graph at runtime. `KodoEntityManagers` expose the following APIs for controlling detached state:

```
public static final int DETACH_LOADED;
public static final int DETACH_FGS;
public static final int DETACH_ALL;
public int getDetachState ();
public void setDetachState (int mode);
```

The JDO `FetchPlan` expose the following APIs for controlling detached state:

```
public static final int DETACH_LOAD_FIELDS;
public static final int DETACH_UNLOAD_FIELDS;
public int getDetachmentOptions ();
public void setDetachmentOptions (int options);
```

The JDO Overview covers these APIs in [Chapter 12, *FetchPlan* \[254\]](#). In addition, the `The KodoFetchPlan` extension adds another option flag for detaching all fields:

```
public static final int DETACH_ALL_FIELDS;
```

11.1.3.1. Detached State Field

When the detached state field is enabled, the Kodo enhancer adds an additional field to the enhanced version of your class. This field of type `Object`. Kodo uses this field for bookkeeping information, such as the versioning data needed to detect optimistic concurrency violations when the object is re-attached.

It is possible to define this detached state field yourself. Declaring this field in your class metadata prevents the enhancer from adding any extra fields to the class, and keeps the enhanced class serialization-compatible with the unenhanced version. The detached state field must not be persistent. See [Section 6.4.1.3, “Detached State” \[484\]](#) for details on how to declare a detached state field.

JPA:

```
import kodo.persistence.*;

@Entity
public class Magazine
    implements Serializable
{
    private String name;
    @DetachedState private Object state;
    ...
}
```

JDO:

```
public class Magazine
    implements Serializable
{
    private String name;
    private Object detachedState;
    ...
}

<?xml version="1.0"?>
<jdo>
  <package name="org.mag">
    <class name="Magazine" detachable="true">
      <extension vendor-name="kodo" key="detached-state-field"
        value="detachedState"/>
    </class>
  </package>
</jdo>
```

11.1.4. Automatic Detachment

The JPA Overview describes JPA's automatic detach behavior in [Section 7.3, “Persistence Context” \[67\]](#). We describe Kodo's options for automatic detach in JDO below.

Detachable JDO objects automatically detach when they are serialized. Kodo expands this automatic detach behavior with optional automatic detachment on various events: close, commit, and non-transactional read. On each configured event, all managed ob-

jects in the `PersistenceManager` cache become detached. Non-detachable objects become transient.

- `close`: Detach all objects when the `PersistenceManager` closes.
- `commit`: Detach all objects when a transaction ends.
- `nontx-read`: Reads outside of a transaction will automatically detach instances before returning them.

By using the proper set of these options, you can avoid unnecessary `detach` calls in your code. Some common use cases for this functionality are:

- **Client/Server applications with short-lived `PersistenceManagers`.** Clients request objects from the server, who returns detached instances which can be modified and manipulated without a local `PersistenceManager`. Changes can then be re-attached later through the server.
- **Servlets and JSPs.** A very common pattern in servlet and JSP architectures is to allocate a `PersistenceManager` at the beginning of a web request, then close it at the end of the request. You can store detached instances between requests in the HTTP session, to maintain changes to an object over several pages for example.
- **Session Beans.** These beans can return a portion of the object graph which remain valid no matter what tier the bean is being called from.

No automatic detachment is performed in JDO by default. You can turn on automatic detachment at runtime through the `KodoPersistenceManager.setDetachAllOn*` methods. You can also set detach detachment triggers with the `kodo.AutoDetach` configuration property:

```
kodo.AutoDetach: close, commit, nontx-read
```

11.2. Remote Managers

In Kodo, each factory maintains a set of resources shared by all the `EntityManagers` or `PersistenceManagers` produced by that factory. Sharing common structures like connection pools and data caches drastically reduces the resource consumption of each manager, increasing your application's scalability. Kodo takes this concept one step further by also giving its factories the ability to act as servers for `PersistenceManagers` or `EntityManagers` on remote machines. You can thus leverage the full JDO or Java Persistence API in your client tier without duplicating limited resources like database connections on each client.

In addition to ensuring that your application scales as more clients are added, this model may allow you to use Kodo in situations where the client machine cannot directly access the necessary server-side resources itself - for example, when the database is only available to the local network.

Important

This feature requires **Kodo Enterprise Edition**.

This remote capability means that you can design your application as a simple two-tiered servlet-database application, and then migrate to a more scalable servlet-Kodo middle tier-database architecture as the load on your system increases. This end picture looks much like a standard J2EE application server architecture, except that the code that uses the persistence APIs in the servlet does not need to change at all to toggle between the more performant two-tier architecture and the more scalable three-tier architecture.

Additionally, Kodo's remote capability is useful for applet and Java Web Start application development. In conjunction with the compressed HTTP transport, you can deploy code that uses standard persistence APIs in an applet or a Web Start application.

The `EntityManager`s and `PersistenceManager`s in these applications will then connect back to the server that they were downloaded from in order to access the database.

11.2.1. Standalone Persistence Server

To configure a factory act as a standalone server to remote clients, the factory's `kodo.PersistenceServer` configuration property to a plugin string (see [Section 2.5, "Plugin Configuration" \[395\]](#)) describing the `com.solarmetric.remote.Transport` implementation to use for remote communication. You can implement your own `Transport`, or use one of the built-in options:

- `false`: The default value. No server is started.
- `tcp`: An alias for `com.solarmetric.remote.TCPTransport`, a TCP transport layer. This transport layer has the following settings:
 - `Port`: The port the server will listen on. Defaults to 5637.
 - `Host`: The host name of the server. Defaults to `localhost`. This setting is not used by the server, but by clients. We discuss client configuration below.
 - `SoTimeout`: The socket read timeout in milliseconds. Defaults to 0 for no timeout.
 - `Decorators`: See [Section 11.2.4, "Data Compression and Filtering" \[586\]](#)

Example 11.2. Configuring a Standalone Persistence Server

JPA XML format:

```
<property name="kodo.PersistenceServer" value="tcp(Port=5555)"/>
```

JDO properties format:

```
kodo.PersistenceServer: tcp(Port=5555)
```

When a factory is configured as a standalone server, it begins listening for client connections automatically as soon as you obtain the factory. The server will run for as long as the factory remains open, and will shut down when you close the factory.

The `KodoEntityManagerFactory` Javadoc details the methods Kodo exposes for manually managing an `EntityManagerFactory`'s server thread.

The `KodoPersistenceManagerFactory` Javadoc details the methods Kodo exposes for manually managing a `PersistenceManagerFactory`'s server thread.

Your Kodo distribution includes a program to start a standalone persistence server. You can run the program through its Java

class, `kodo.jdbc.kernel.StartPersistenceServer`, or through the provided `startserver` command-line script. The script accepts the standard Kodo command-line arguments outlined in [Section 2.4, “Command Line Configuration” \[394\]](#).

Example 11.3. Starting a Standalone Persistence Server

```
startserver -p server.properties
```

11.2.2. HTTP Persistence Server

Kodo's remote managers can communicate with the server over HTTP, allowing you to use them through firewalls that shut off other ports and protocols. In order to receive HTTP requests from remote clients, Kodo includes the `kodo.remote.PersistenceServerServlet`. You can deploy this servlet into any standards-compliant servlet container.

The `PersistenceServerServlet` services remote requests using an internal factory. The servlet provides several mechanisms for configuring this factory:

1. First, the servlet checks the value of the `kodo.jndi` servlet initialization parameter (servlet initialization parameters are specified in the standard `web.xml` deployment file; see your servlet container documentation for details). If the value of this parameter is non-null, Kodo attempts to look up the `kodo.kernel.BrokerFactory` at the indicated JNDI location. (A `BrokerFactory` is the native Kodo component that underlies every `PersistenceManagerFactory` or `EntityManagerFactory`.)
2. If the `kodo.jndi` initialization parameter is not set, Kodo checks the `kodo.properties` initialization parameter. The value of this parameter is a resource path to an JPA XML or a JDO properties file containing Kodo configuration properties.
3. Finally, Kodo checks the remainder of the servlet initialization parameters for Kodo configuration properties. These parameter values override the value supplied in the configuration file (if any).

If you use servlet parameters alone to configure the persistence server, Kodo will not know whether to apply JPA or JDO defaults. Set the `kodo.Specification` servlet parameter to `ejb` or `jdo` to tell Kodo which specification defaults to apply.

You can make sure that the servlet's factory is configured as expected by navigating to the servlet in your web browser. The servlet will display a simple web page detailing the configuration of its internal factory.

11.2.3. Client Managers

Client `EntityManagers` and `PersistenceManagers` are remote proxies to server-side managers created by the server-side factory you are communicating with. From an API standpoint, a client manager is exactly like a local one, complete with all Kodo API extensions. Behind the scenes, however, the actions you take on a client manager are sent to the corresponding server-side manager for processing. For performance reasons and because your persistent objects are not proxies themselves, each client manager has a local cache of managed objects, synchronized with the server-side manager's cache.

You obtain client `EntityManagers` and `PersistenceManagers` in the same way you obtain local managers: from an `EntityManagerFactory` or `PersistenceManagerFactory` that you have constructed through JCA or the `PersistenceHelper` / `JDOHelper` helper. Client configuration properties are the same as those used for local Kodo operation, with the following exceptions:

- You must set the **kodo.BrokerFactory** property to `remote`.
- The **kodo.PersistenceServer** setting is used to find the remote server. If you are using a **standalone server**, the value of this property is typically the same as its value on the server. If you are using the **HTTP servlet**, the value of this property on the client is:

```
http(URL=<servlet-url>)
```

`http` in the setting above is an alias for the **com.solarmetric.remote.HTTPTransport**, whose `URL` property indicates the URL to connect to. You can also specify a `Decorators` property, as discussed in **Section 11.2.4, “Data Compression and Filtering” [586]**

- The **kodo.ConnectionRetainMode** property controls how the client handles connections to the server, not how the server handles connections to the database. The available values are the same as the options for database connections:
 - `always`: Each client manager obtains a single connection to its server-side counterpart and uses this connection until it is closed.
 - `transaction`: A connection is obtained when each transaction begins, and relinquished when the transaction completes. Nontransactional connections are obtained as needed and released immediately.
 - `on-demand`: A connection to the server is obtained when needed, and immediately closed when the request has been fulfilled. This is the default.
- **kodo.ConnectionFactoryProperties** controls pooling not for database connections, but for connections from the client machine to the remote server. The following pooling options are available:
 - `ExceptionAction`: The action to take when when a connection that has thrown an exception is returned to the pool. Set to `destroy` to destroy the connection. Set to `validate` to validate the connection (subject to the `TestOnReturn`, `TestOnBorrow`, and other test settings). Set to `none` to ignore the fact that the connection has thrown an exception, and assume it is still usable. Defaults to `destroy`.
 - `MaxActive`: The maximum number of connections in use at one time. Defaults to 8.
 - `MaxIdle`: The maximum number of idle connections to keep in the pool. Defaults to 8.
 - `MaxWait`: The maximum number of milliseconds to wait for a free connection to become available before giving up. Defaults to 3000.
 - `MinEvictableIdleTimeMillis`: The minimum number of milliseconds that a connection can sit idle before it becomes a candidate for eviction from the pool. Defaults to 30 minutes. Set to 0 to never evict a connection based on idle time alone.
 - `TestOnBorrow`: Whether to validate connections before obtaining them from the pool. Defaults to `true`.
 - `TestOnReturn`: Set to `true` to validate connections when they are returned to the pool.
 - `TestWhileIdle`: Set to `true` to periodically validate idle connections.
 - `TimeBetweenEvictionRunsMillis`: The number of milliseconds between runs of the eviction thread. Defaults to -1, meaning the eviction thread will never run.
 - `ValidationTimeout`: The minimum number of milliseconds that must elapse before a connection will ever be re-validated. This property is usually used with `TestOnBorrow` or `TestOnReturn` to reduce the number of validations performed, because the same connection is often borrowed and returned many times in short periods of time. Defaults to 300000 (5 minutes).

- `WhenExhaustedAction`: The action to take when there are no available connections in the pool. Set to `exception` to immediately throw an exception. Set to `block` to block until a connection is available or the maximum wait time is exceeded. Set to `grow` to automatically grow the pool. Defaults to `block`.

Remember that persistent connections to the server consume server-side resources, and therefore should be minimized if possible. To disable pooling altogether, set `MaxActive` to 0.

- Database connectivity and JDBC-related properties are ignored by the client factory. All database communication takes place on the server, so these properties are only valid on the server-side factory. There are, however, two exceptions to this rule. If specified, the client will transfer your local `kodo.ConnectionUserName` and `kodo.ConnectionPassword` settings to the server. This allows different remote clients to connect as different database users.

Other than the bullet points above, you configure client factories in the same way as local factories. Keep in mind, though, that the configuration you specify on the client only applies to the client factory, not the server. For example, if you configure a data cache and query cache on the client, these caches will only "see" changes made by the client; they will not automatically synchronize with changes made by any other client or changes made on the server. Thus, you will typically want to configure components like the data cache, query cache, lock manager, etc. on the server only (where clients can still benefit from them by proxy), and turn them off on the client.

Example 11.4. Client Configuration

JPA XML format:

```
<property name="kodo.BrokerFactory" value="remote"/>
<property name="kodo.PersistenceServer" value="tcp(Host=kodohost.mydomain.com, Port=5555)"/>
<property name="kodo.ConnectionRetainMode" value="transaction"/>
<property name="kodo.ConnectionFactoryProperties" value="MaxIdle=3, ValidationTimeout=60000"/>
```

JDO properties format:

```
kodo.BrokerFactory: remote
kodo.PersistenceServer: tcp(Host=kodohost.mydomain.com, Port=5555)
kodo.ConnectionRetainMode: transaction
kodo.ConnectionFactoryProperties: MaxIdle=3, ValidationTimeout=60000
```

Example 11.5. HTTP Client Configuration

JPA XML format:

```
<property name="kodo.BrokerFactory" value="remote"/>
<property name="kodo.PersistenceServer" value="http(URL=http://jdohost.mydomain.com/tomcat/pmsserver)"/>
<property name="kodo.ConnectionRetainMode" value="transaction"/>
<property name="kodo.ConnectionFactoryProperties" value="MaxIdle=3, ValidationTimeout=60000"/>
```

JDO properties format:

```
kodo.BrokerFactory: remote
kodo.PersistenceServer: http(URL=http://jdohost.mydomain.com/tomcat/pmserver)
kodo.ConnectionRetainMode: transaction
kodo.ConnectionFactoryProperties: MaxIdle=3, ValidationTimeout=60000
```

11.2.4. Data Compression and Filtering

Kodo's built in transport implementations - `tcp`, `http` - allow you to wrap their data streams in decorators to add additional functionality such as data compression and filtering. Each accepts a `Decorators` configuration property specifying a semi-colon-separated list of `com.solarmetric.remote.StreamDecorators` to decorate the input and output streams between the client and server. Each item in the list can be the full class name of a custom decorator, or one of the following built-in aliases:

- `gzip`: Use gzip compression when transferring data.

Example 11.6. Enabling Compression with the TCP Transport

On the server, set the `kodo.PersistenceServer` property to:

```
tcp(Port=5555, Decorators=gzip)
```

And on the client:

```
tcp(Host=jdohost.mydomain.com, Port=5555, Decorators=gzip)
```

Example 11.7. Enabling Compression with the HTTP Transport

In the server properties file / servlet configuration, set the `kodo.PersistenceServer` property to:

```
http(Decorators=gzip)
```

And on the client:

```
http(URL=http://jdohost.mydomain.com/tomcat/pmserver, Decorators=gzip)
```

11.2.5. Remote Persistence Deployment

Using Kodo's remote features involves deploying Kodo to the server machine as well as one or more client machines. Deploying Kodo on the server is exactly the same as deploying Kodo for local use. You must include all Kodo libraries, your configuration properties file (if you use one), your logging configuration file (again, if you use one), your JDBC drivers, your enhanced persistent classes, your metadata, and your O/R mapping information. All of these topics are covered in other sections of this manual.

Deploying Kodo on the client is also the same as deploying Kodo for local use, with two small exceptions:

1. JDBC libraries are not required on the client.
2. O/R mapping information is not required on the client.

Note that you may include the above information in your deployment; it is simply not required.

11.2.6. Remote Transfer Listeners

The Kodo remote package provides a mechanism for your application to register an object that can listen for transfer events. Transfer events take place during flush operations as objects are sent to the server, and when objects are loaded from the server by extents, queries, or obtaining objects by id. For more details, please see the javadoc for `kodo.remote.RemoteTransferListener`.

11.3. Remote Event Notification Framework

The remote event notification framework allows a subset of the information available through Kodo's transaction events (see [Section 9.8, "Transaction Events" \[559\]](#)) to be broadcast to remote listeners. Kodo's **data cache**, for example, uses remote events to remain synchronized when deployed in multiple JVMs.

Important

This feature requires **Kodo Enterprise Edition** or **Kodo Standard Edition** with the **Performance Pack**.

To enable remote events, you must configure the `EntityManagerFactory` or `PersistenceManagerFactory` to use a `RemoteCommitProvider` (see below).

When a `RemoteCommitProvider` is properly configured, you can register **RemoteCommitListeners** that will be alerted with a list of modified object ids whenever a transaction on a remote machine successfully commits.

11.3.1. Remote Commit Provider Configuration

Kodo includes built in remote commit providers for JMS and TCP communication.

11.3.1.1. JMS

Kodo includes built in remote commit providers for JMS and TCP communication. The JMS remote commit provider can be configured by setting the `kodo.RemoteCommitProvider` property to contain the appropriate configuration properties. The JMS provider understands the following properties:

- `Topic`: The topic that the remote commit provider should publish notifications to and subscribe to for notifications sent from other JVMs. Defaults to `topic/KodoCommitProviderTopic`
- `TopicConnectionFactory`: The JNDI name of a `javax.jms.TopicConnectionFactory` factory to use for

finding topics. Defaults to `java:/ConnectionFactory`. This setting may vary depending on the application server in use; consult the application server's documentation for details of the default JNDI name for the `javax.jms.TopicConnectionFactory` instance. For example, under Weblogic, the JNDI name for the `TopicConnectionFactory` is `javax.jms.TopicConnectionFactory`.

- `ExceptionReconnectAttempts`: The number of times to attempt to reconnect if the JMS system notifies Kodo of a serious connection error. Defaults to 0, meaning Kodo will log the error but otherwise ignore it, hoping the connection is still valid.
- `*`: All other configuration properties will be interpreted as settings to pass to the JNDI `InitialContext` on construction. For example, you might set the `java.naming.provider.url` property to the URL of the context provider.

To configure a factory to use the JMS provider, your properties might look like the following:

Example 11.8. JMS Remote Commit Provider Configuration

JPA XML format:

```
<property name="kodo.RemoteCommitProvider" value="jms(Topic=topic/KodoCommitProviderTopic)" />
```

JDO properties format:

```
kodo.RemoteCommitProvider: jms(Topic=topic/KodoCommitProviderTopic)
```

Note

Because of the nature of JMS, it is important that you invoke `EntityManagerFactory.close` or `PersistenceManagerFactory.close` when finished with a factory. If you do not do so, a daemon thread will stay up in the JVM, preventing the JVM from exiting.

11.3.1.2. TCP

The TCP remote commit provider has several options that are defined as host specifications containing a host name or IP address and an optional port separated by a colon. For example, the host specification `saturn.bea.com:1234` represents an `InetAddress` retrieved by invoking `InetAddress.getByName("saturn.bea.com")` and a port of 1234.

The TCP provider can be configured by setting the `kodo.RemoteCommitProvider` plugin property to contain the appropriate configuration settings. The TCP provider understands the following properties:

- `Port`: The TCP port that the provider should listen on for commit notifications. Defaults to 5636.
- `Addresses`: A semicolon-separated list of IP addresses to which notifications should be sent. No default value.
- `NumBroadcastThreads`: The number of threads to create for the purpose of transmitting events to peers. You should increase this value as the number of concurrent transactions increases. The maximum number of concurrent transactions is a function of the size of the connection pool. See the `MaxActive` property of

`kodo.ConnectionFactoryProperties` in **Section 4.1, “Using the Kodo DataSource” [423]**. Setting a value of 0 will result in behavior where the thread invoking `commit` will perform the broadcast directly. Defaults to 2.

- `RecoveryTimeMillis`: Amount of time to wait in milliseconds before attempting to reconnect to a peer of the cluster when connectivity to the peer is lost. Defaults to 15000.
- `MaxIdle`: The number of TCP sockets (channels) to keep open to each peer in the cluster for the transmission of events. Defaults to 2.
- `MaxActive`: The maximum allowed number of TCP sockets (channels) to open simultaneously between each peer in the cluster. Defaults to 2.

To configure a factory to use the TCP provider, your properties might look like the following:

Example 11.9. TCP Remote Commit Provider Configuration

JPA XML format:

```
<property name="kodo.RemoteCommitProvider"
  value="tcp(Addresses=10.0.1.10;10.0.1.11;10.0.1.12;10.0.1.13)"/>
```

JDO properties format:

```
kodo.RemoteCommitProvider: tcp(Addresses=10.0.1.10;10.0.1.11;10.0.1.12;10.0.1.13)
```

11.3.1.3. Common Properties

In addition to the provider-specific configuration options above, all providers accept the following plugin properties:

- `TransmitPersistedObjectIds`: Whether remote commit events will include the object ids of instances persisted in the transaction. By default only the class names of types persisted in the transaction are sent. This results in smaller events and more efficient network utilization. If you have registered your own remote commit listeners, however, you may require the persisted object ids as well.

To transmit persisted object ids in our remote commit events using the JMS provider, we modify the previous example as follows:

Example 11.10. Transmitting Persisted Object Ids

JPA XML format:

```
<property name="kodo.RemoteCommitProvider"
  value="jms(Topic=topic/KodoCommitProviderTopic, TransmitPersistedObjectIds=true)"/>
```

JDO properties format:

```
kodo.RemoteCommitProvider: jms(Topic=topic/KodoCommitProviderTopic, TransmitPersistedObjectIds=true)
```

11.3.2. Customization

You can develop additional mechanisms for remote event notification by creating an implementation of the **RemoteCommitProvider** interface, possibly by extending the **AbstractRemoteCommitProvider** abstract class. For details on particular customization needs, contact us at support@solarmetric.com.

Chapter 12. Management and Monitoring

The management and monitoring capability uses the Java Management Extensions (JMX) standard to allow for both local and remote management of key:

- Attributes
- Operations
- Notifications / Performance Statistics

Important

This feature requires **Kodo Enterprise Edition** or **Kodo Standard Edition** with the **Performance Pack**.

The unit of management in JMX is the Managed Bean (MBean). Kodo provides a number of MBeans that allow management of key components. For example, the connection pool MBean provides statistics on the numbers of active and idle connections, and allows modification of the attributes controlling the maximum number of active connections.

The default JMX implementation used by Kodo is MX4J version 1.1.1 (see mx4j.sourceforge.net). The MX4J jar, `mx4j-jmx.jar`, is included in the Kodo distribution, and is covered under the MX4J license, which can be found in the `lib` directory. To use a different implementation, simply remove the `mx4j-jmx.jar` file from your `CLASSPATH` and insert the appropriate jars for your JMX implementation.

Note that the resource archive `kodo.rar` provided with Kodo does not include MX4J. This is because most application servers provide their own JMX implementation. However, a small subset of the MX4J implementation is included.

The management and monitoring capability can be run both locally and remotely. An example of its use can be found in the `management` sample described in [Section 1.2.2, “JMX Management” \[619\]](#)

12.1. Configuration

The Management capability is configured via the standard Kodo configuration system using the `kodo.ManagementConfiguration` property. This property is a plugin string (see [Section 2.5, “Plugin Configuration” \[395\]](#)), so you can also set it to the full class name of a custom `ManagementConfiguration`. Pre-defined values are:

- `none`: No management or profiling turned on. This is the default.
- `profiling-gui`: Turn on the local profiling GUI (see [Section 13.1, “Profiling in an Embedded GUI” \[603\]](#) for more configuration information).
- `profiling-export`: Export profiling data (see [Section 13.2, “Dumping Profiling Data to Disk from a Batch Process” \[605\]](#) for more configuration information).
- `profiling`: Enable profiling without export or GUI. Useful when trying to access the `ProfilingAgent` programmatically.
- `mgmt`: Enable management. Suitable for use in JBoss and other environments where all MBeans should be registered with a JMX `MBeanServer` for either management via a user defined mechanism, or via a mechanism defined by the `MBeanServer`. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)).
- `mgmt-prof`: Enable management, including profiling. Suitable for use in JBoss and other environments where all MBeans

should be registered with a JMX MBeanServer for either management via a user defined mechanism, or via a mechanism defined by the MBeanServer. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#)

- `mgmt-export`: Enable management, and enable a profiling export. Suitable for use in JBoss and other environments where all MBeans should be registered with a JMX MBeanServer for either management via a user defined mechanism, or via a mechanism defined by the MBeanServer. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#) and [Section 13.2, “Dumping Profiling Data to Disk from a Batch Process” \[605\]](#)
- `mx4j1-remote-mgmt`: Enable remote management via MX4J v.1.x implementations (supporting versions of the JMX specification prior to 1.2). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) and the `Remote Group` (see [Section 12.1.2, “Optional Parameters in Remote Group” \[594\]](#))
- `mx4j1-remote-mgmt-prof`: Enable remote management, including remote profiling via MX4J v.1.x implementations (supporting versions of the JMX specification prior to 1.2). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) and the `Remote Group` (see [Section 12.1.2, “Optional Parameters in Remote Group” \[594\]](#)) Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#)
- `mx4j1-remote-mgmt-export`: Enable remote management, and enable a profiling export via MX4J v.1.x implementations (supporting versions of the JMX specification prior to 1.2). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) and the `Remote Group` (see [Section 12.1.2, “Optional Parameters in Remote Group” \[594\]](#)) Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#) and [Section 13.2, “Dumping Profiling Data to Disk from a Batch Process” \[605\]](#)
- `jmx2-remote-mgmt`: Enable remote management via JMX v.1.2 implementations (supporting JSR 160 for remote management). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) and the `JSR 160 Group` (see [Section 12.1.3, “Optional Parameters in JSR 160 Group” \[594\]](#))
- `jmx2-remote-mgmt-prof`: Enable remote management, including remote profiling via JMX v.1.2 implementations (supporting JSR 160 for remote management). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) and the `JSR 160 Group` (see [Section 12.1.3, “Optional Parameters in JSR 160 Group” \[594\]](#)) Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#)
- `jmx2-remote-mgmt-export`: Enable remote management, and enable a profiling export via JMX v.1.2 implementations (supporting JSR 160 for remote management). Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) and the `JSR 160 Group` (see [Section 12.1.3, “Optional Parameters in JSR 160 Group” \[594\]](#)) Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#) and [Section 13.2, “Dumping Profiling Data to Disk from a Batch Process” \[605\]](#)
- `local-mgmt`: Enable management, bringing up the JMX management console in the local JVM. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#))
- `local-mgmt-prof`: Enable management, including profiling, bringing up the JMX management console in the local JVM. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#)
- `local-mgmt-export`: Enable management, bringing up the JMX management console in the local JVM, and enable a profiling export. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#) and [Section 13.2, “Dumping Profiling Data to Disk from a Batch Process” \[605\]](#)

- `wl81-mgmt`: Enable WebLogic 8.1 management. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) and the `WebLogic 8.1 Group` (see [Section 12.1.4, “Optional Parameters in WebLogic 8.1 Group” \[595\]](#))
- `wl81-mgmt-prof`: Enable WebLogic 8.1 management, including remote profiling. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) and the `WebLogic 8.1 Group` (see [Section 12.1.4, “Optional Parameters in WebLogic 8.1 Group” \[595\]](#)). Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#)
- `wl81-mgmt-export`: Enable WebLogic 8.1 management, and enable a profiling export. Supports optional parameters in the `Management Group` (see [Section 12.1.1, “Optional Parameters in Management Group” \[593\]](#)) and the `WebLogic 8.1 Group` (see [Section 12.1.4, “Optional Parameters in WebLogic 8.1 Group” \[595\]](#)). Also supports optional parameters described in [Section 13.3, “Controlling How the Profiler Obtains Context Information” \[606\]](#) and [Section 13.2, “Dumping Profiling Data to Disk from a Batch Process” \[605\]](#)

12.1.1. Optional Parameters in Management Group

Those `kodo.ManagementConfiguration` plugins that provide JMX based management have three optional parameters, described in this section.

If JMX based management is desired, Kodo needs to either find an existing `MBeanServer` or create a new one. Appropriate plugins for the configuration property `kodo.ManagementConfiguration` have an optional parameter `MBeanServerStrategy` that allows for configuration of this process. The following options are available:

- `any-create`: Attempt to find an existing `MBeanServer`. If multiple are found, use the first one. If none are found, create a new server. This is the default.
- `create`: Do not attempt to find an existing `MBeanServer`. Always create a new server.
- `agentId:<agent id>`: Attempt to find an existing `MBeanServer` with the given agent id. If such a server is not found, create a new server.

For example, in order to force creation of a new `MBeanServer` when doing remote management, include the following line in your configuration:

JPA XML format:

```
<property name="kodo.ManagementConfiguration" value="remote-mgmt(MBeanServerStrategy=create)"/>
```

JDO properties format:

```
kodo.ManagementConfiguration: remote-mgmt(MBeanServerStrategy=create)
```

The other optional parameters are: `EnableLogMBean` and `EnableRuntimeMBean`.

- `EnableLogMBean`: When set to true, indicates that the `LogMBean` should be registered.
- `EnableRuntimeMBean`: When set to true, indicates that the `RuntimeMBean` should be registered.

For example, to register both the `RuntimeMBean` and the `LogMBean`, set `kodo.ManagementConfiguration` to:

```
local-mgmt(EnableLogMBean=true,EnableRuntimeMBean=true)
```

12.1.2. Optional Parameters in Remote Group

In order to do remote management, you need to start a remote JMX adaptor. Once the adaptor has started, you need to start the GUI. See [Section 12.2, “Kodo Management Console” \[595\]](#) for more information. The `Remote Group` enables configuration of the remote management adaptor. For adaptors other than the `mx4j-jrmp` adaptor, a custom `ManagementConfiguration` must be defined. By default, the `mx4j-jrmp` adaptor sets up an RMI registry naming service on `rmi://localhost:1099` and registers an MX4J JRMP JMX adaptor under the JNDI name `jrmp`. To change the host, port and JNDI name used, the following optional parameters are available on those `kodo.ManagementConfiguration` plugins that support the `Remote Group`.

- `JNDIName`: The JNDI name under which to register the remote JMX adaptor. Defaults to `jrmp`.
- `Host`: The hostname on which the RMI registry naming service will listen. Defaults to `localhost`.
- `Port`: The port on which the RMI registry naming service will listen. Defaults to `1099`.

For example, to listen on port `2345`, set `kodo.ManagementConfiguration` to:

```
remote-mgmt(Port=2345)
```

12.1.3. Optional Parameters in JSR 160 Group

In order to do remote management using the JSR 160 standard (supported by JMX implementations supporting JMX 1.2 and higher), you need to start a remote JMX adaptor. Once the adaptor has started, you need to start the GUI. See [Section 12.2, “Kodo Management Console” \[595\]](#) for more information. The `JSR 160 Group` enables configuration of the remote management adaptor.

For convenience, the JSR 160 adaptor sets up an RMI registry naming service on `rmi://localhost:1099` and registers the JMX Connector Server with it, by default. To change the host, port and service URL used, the following optional parameters are available on those `kodo.ManagementConfiguration` plugins that support the `JSR 160 Group`.

- `ServiceURL`: The JMX service URL name under which to register the JMX Connector Server. Defaults to `service:jmx:rmi://localhost/jndi/jmxservice`, indicating that the RMI connector will be used and registered under the JNDI name `jmxservice`.
- `NamingImpl`: The classname of the naming service implementation to start in order to register the RMI connector with a JNDI name. Defaults to `mx4j.tools.naming.NamingService`, which is appropriate for MX4J v. 2.x. If set to the empty string, no naming service will be started. This is appropriate if a naming service is already running, or if a non-RMI connector is used.
- `Host`: The hostname on which the RMI registry naming service will listen. Defaults to `localhost`. This parameter is ignored for connectors that do not register with a naming service.
- `Port`: The port on which the RMI registry naming service will listen. Defaults to `1099`. This parameter is ignored for con-

nectors that do not register with a naming service.

For example, to have the RMI registry naming service listen on port 2345, set `kodo.ManagementConfiguration` to:

```
jmx2-remote-mgmt (Port=2345)
```

12.1.4. Optional Parameters in WebLogic 8.1 Group

The following parameters must be set on those `kodo.ManagementConfiguration` plugins that support the WebLogic 8.1 Group. Once WebLogic has started, the GUI needs to be started. See [Section 12.2, “Kodo Management Console” \[595\]](#) for more information. For additional requirements in order to do remote management, please see [Section 12.2.1.1, “Connecting to Kodo under WebLogic 8.1” \[596\]](#)

- `UserName`: The username that Kodo should use to access the WebLogic MBeanServer. This must be set.
- `Password`: The password that Kodo should use to access the WebLogic MBeanServer. This must be set.
- `ServerName`: The server name of the server whose MBeanServer to which Kodo should connect. This must be set.
- `URL`: The URL to which Kodo should connect to access the WebLogic MBeanServer. Defaults to `t3://localhost:7001`.

For example, set `kodo.ManagementConfiguration` to:

```
wl81-mgmt (UserName=admin,Password=admin,ServerName=myserver,URL="t3://localhost:7001")
```

12.1.5. Configuring Logging for Management / Monitoring

Logging for the management and monitoring API is on the `kodo.Manage` logging channel.

12.2. Kodo Management Console

The Kodo Management Console is used for local and remote management of MBeans. It can be used to connect to a local MBean server or multiple remote MBean servers. To connect to a local server, see [Section 12.1, “Configuration” \[591\]](#).

12.2.1. Remote Connection

To start the Kodo Management Console for remote management, run the `remotejmxtool` command. The `remotejmxtool` accepts the following arguments:

- `-connect/-c`: Whether to attempt an initial connection to the remote JMX adaptor. Defaults to false.
- `-type/-t`: The type of the remote JMX adaptor. Current supported types are `mx4j1`, `jmx2`, `weblogic81` and `jboss`. Defaults to `mx4j1`. Integration with other JMX server implementations that support remote connectivity can be accomplished by creating a class that implements the `RemoteMBeanServerFactory` interface. In this case, the `type` should be the fully qualified name of the implementing class.

- `-host / -h`: Hostname of the JNDI service provider where the remote JMX adaptor is registered. Defaults to `localhost`. When attempting an initial connection to WebLogic, this must be set to a hostname of the form `username:password@hostname`. This is optional for JSR 160 connectors, as it may not be necessary for some connectors, and may be encoded in the JMX service URL for others.
- `-port / -p`: Port of the JNDI service provider where the remote JMX adaptor is registered. Defaults to 1099 when connecting to MX4J. Defaults to 7001 when connecting to WebLogic. This is optional for JSR 160 connectors, as it may not be necessary for some connectors, and may be encoded in the JMX service URL for others.
- `-name / -n`: For non-JSR 160 connectors, the JNDI name of the remote JMX adaptor. Defaults to a special value `default` which yields the default JNDI name appropriate for the chosen remote JMX adaptor type. For MX4J, the default is `jrmp`, and for JBoss, the default is the first available JMX adaptor at the specified JNDI service provider. For WebLogic, this parameter is ignored. For JSR 160 connectors, this is the JMX service URL, and defaults to `service:jmx:rmi:///localhost/jndi/jmxservice`. Note that this can also encode the `host` and `port` parameters, if desired. For example, the default JMX Connector Server could be referenced by `service:jmx:rmi:///localhost/jndi/rmi:///localhost:1099/jmxservice`. In that case, the `Host` and `Port` parameters will be ignored.

For example, to automatically connect to the MX4J remote JMX adaptor on host `myhost.mydomain.com`, use the following command:

```
remotejmxtool -c true -host myhost.mydomain.com
```

Once `remotejmxtool` is up, you can connect to multiple remote JMX adaptors.

To connect to Kodo with MX4J v. 1.1.x, select `Connect to Kodo JMX...` from the File menu.

To connect to Kodo with a JSR 160 connector, select `Connect to Kodo JMX 1.2...` from the File menu.

To connect to Kodo running under WebLogic, select `Connect to Kodo via WebLogic JMX...` from the File menu.

To connect to Kodo running under JBoss, select `Connect to Kodo via JBossMX...` from the File menu.

12.2.1.1. Connecting to Kodo under WebLogic 8.1

In order to connect to WebLogic 8.1 with `remotejmxtool`, the following requirements must be met:

- `remotejmxtool` must be run with the `weblogic.jar` (found in the `weblogic81/server/lib/` directory of the WebLogic 8.1 distribution) in your `CLASSPATH`. Note that this library should appear *before* the `mx4j-jmx.jar` (included with the Kodo distribution) library in your `CLASSPATH`.
- The `remotejmxtool` must be run with JDK 1.4.x.
- The jar `kodo-wl81manage.jar` must be put in the WebLogic system `CLASSPATH`. You can accomplish this by editing `startWebLogic.sh/startWebLogic.cmd`.

12.2.1.2. Connecting to Kodo under JBoss 3.2

In order to connect to JBossMX 3.2, `remotejmxtool` must be run with the following libraries from the JBoss distribution in your `CLASSPATH`.

- `jboss-common-client.jar`: Found in the `client/` directory of the JBoss 3.2 distribution.

- `jboss-jmx.jar`: Found in the `lib/` directory of the JBoss 3.2 distribution.
- `jmx-adaptor-plugin.jar`: Found in the `server/all/lib/` directory of the JBoss 3.2 distribution.
- `jnp-client.jar`: Found in the `client/` directory of the JBoss 3.2 distribution.
- `jboss-system.jar`: Found in the `lib/` directory of the JBoss 3.2 distribution.
- `jnet.jar`: Found in the `client/` directory of the JBoss 3.2 distribution. Alternately, `remotejmxtool` can be run under JDK 1.4 or higher.
- `concurrent.jar`: Found in the `client/` directory of the JBoss 3.2 distribution.
- `jbossall-client.jar`: Found in the `client/` directory of the JBoss 3.2 distribution.

Note that these libraries should appear *before* the `mx4j-jmx.jar` (included with the Kodo distribution) library in your `CLASSPATH`.

12.2.1.3. Connecting to Kodo under JBoss 4

In order to connect to JBossMX 4, `remotejmxtool` must be run with the following libraries from the JBoss distribution in your `CLASSPATH`.

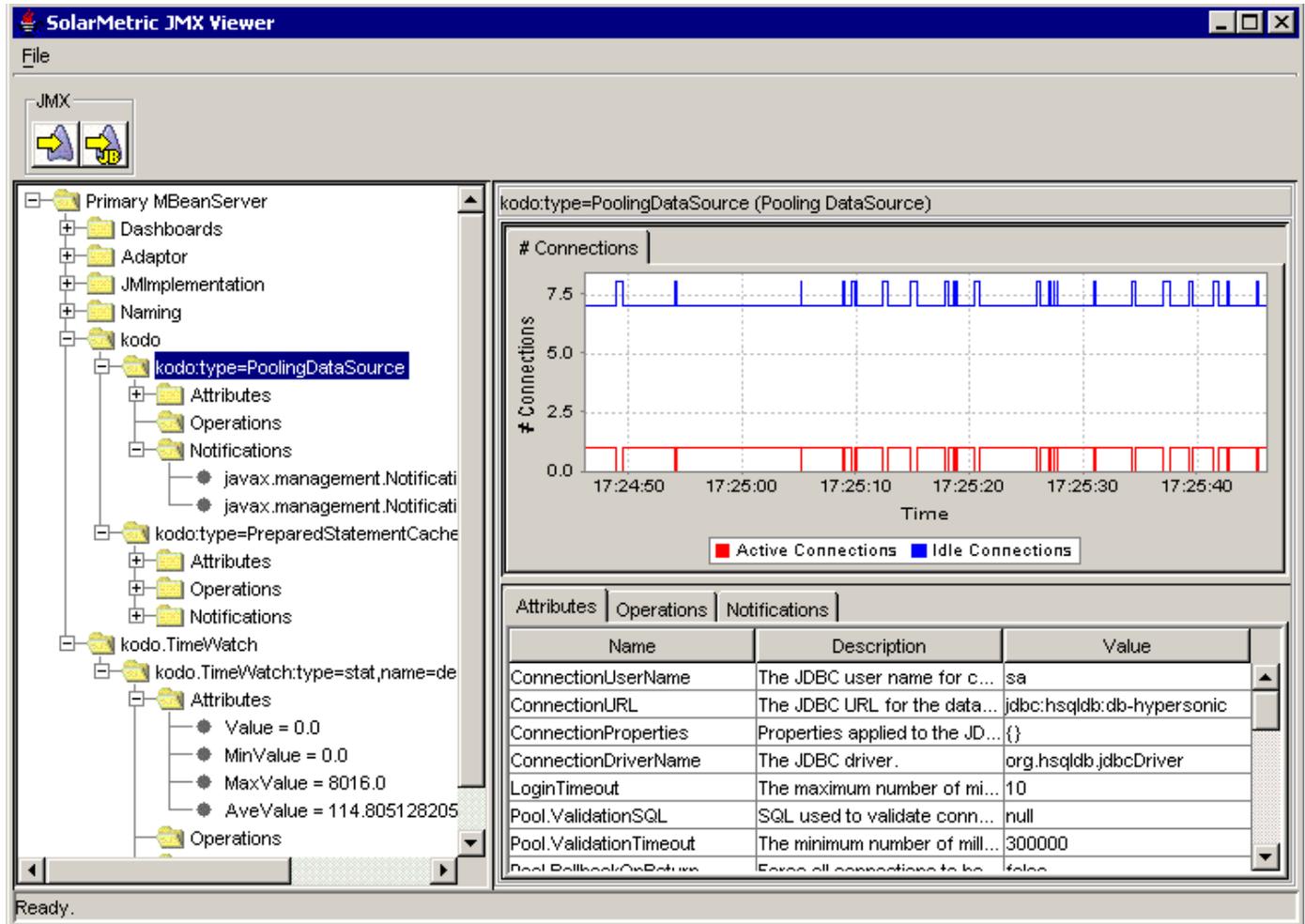
- `jboss-common-client.jar`: Found in the `client/` directory of the JBoss 4 distribution.
- `jboss-jmx.jar`: Found in the `lib/` directory of the JBoss 4 distribution.
- `jmx-adaptor-plugin.jar`: Found in the `server/all/lib/` directory of the JBoss 4 distribution.
- `jnp-client.jar`: Found in the `client/` directory of the JBoss 4 distribution.
- `jboss-system.jar`: Found in the `lib/` directory of the JBoss 4 distribution.
- `concurrent.jar`: Found in the `client/` directory of the JBoss 4 distribution.
- `jbossall-client.jar`: Found in the `client/` directory of the JBoss 4 distribution.
- `dom4j.jar`: Found in the `lib/` directory of the JBoss 4 distribution.

Note that these libraries should appear *before* the `mx4j-jmx.jar` (included with the Kodo distribution) library in your `CLASSPATH`.

Additionally, the following requirements must be met:

- The `remotejmxtool` must be run with JDK 1.5.x.
- The jar `kodo-jboss4manage.jar` must be put in the JBoss 4 system `CLASSPATH`. You can accomplish this by placing the jar in the server's `lib/` directory (e.g. `>JBoss 4 install</server/default/lib/`).

12.2.2. Using the Kodo Management Console



The above diagram shows the Kodo Management Console window. The Kodo Management Console window is divided into two main parts, the JMX Explorer on the left, and the MBean Panel on the right.

12.2.2.1. JMX Explorer

The JMX Explorer provides a tree view of the connected MBean servers. Under each MBean server are the JMX domains handled by that server. Under each domain are the MBeans within that domain. Under each MBean are the *attributes*, *operations* and *notifications* provided by that MBean.

12.2.2.1.1. Executing Operations

In order to execute an operation of an MBean, right click on the operation, and select "Execute..." from the context menu. A dialog box will come up asking for values for each of the arguments to the managed operation. Fill in each of the values and hit the OK button to execute the operation.

Note

Currently, only primitive types, primitive wrapper types, and classes with a string constructor can be entered.

If the operation returns a non-null value, the string representation of the return value is shown.

12.2.2.1.2. Listening to Notifications

When an MBean is selected in the JMX Explorer, the Kodo Management Console automatically listens to all notifications. To stop listening to all notifications for a given MBean, right click on the `Notifications` node and select `Stop Listening All`. To stop listening to a single notification, right click on the individual notification and select `Stop Listening`. In order to listen to all notifications provided by an MBean, right click on the `Notifications` node under the MBean and select `Listen All`. To listen to a single notification, right click on the individual notification and select `Listen`.

You can see the available notifications in the MBean Panel to the right of the JMX Explorer.

12.2.2.2. MBean Panel

You can view the attributes, operations and notifications of an MBean in the MBean Panel. The top half of the panel shows notifications and statistics, while the bottom half allows for viewing / editing attributes, viewing available operations, and viewing available notifications.

12.2.2.2.1. Notifications / Statistics

The top half of the MBean Panel shows the notifications emitted by the selected MBean. Note that you must listen to a notification (see [Section 12.2.2.1.2, “Listening to Notifications” \[598\]](#)) in order to view it in the MBean Panel. There is one tab per notification. Certain notifications represent statistics. These notifications are grouped under tabs based on their ordinate description. Statistic notifications are represented in charts. Dragging a rectangle across a chart causes the chart to zoom in on the selected area. Right clicking on a chart brings up a context menu with a number of options:

- `Properties...`: Edit chart properties, such as colors and labels.
- `Save as...`: Save the chart to disk.
- `Print...`: Print the chart.
- `Zoom In / Zoom Out`: Zoom in and out on either or both axes.
- `Auto Range`: Set the either or both the abscissa and ordinate range to see all of the values.

12.2.2.2.2. Setting Attributes

The `Attributes` tab in the bottom half of the MBean Panel allows for viewing / editing of attributes. Not all attributes are editable. Selecting an editable attribute allows you to set the value.

Note

Currently, you can only enter primitive types, primitive wrapper types, and classes with a string constructor.

12.3. Accessing the MBeanServer from Code

You can access the MBeanServer in which the Kodo MBeans are registered using the `KodoConfiguration` interface's `getMBeanServer` method.

Example 12.1. Accessing the MBeanServer

JPA:

```
KodoEntityManagerFactory kemf = KodoPersistence.cast (emf);
MBeanServer mbServer = kemf.getConfiguration ().getMBeanServer ();
```

JDO:

```
KodoPersistenceManagerFactory kpmf = KodoJDOHelper.cast (pmf);
MBeanServer mbServer = kpmf.getConfiguration ().getMBeanServer ();
```

12.4. MBeans

12.4.1. Log MBean

The Log MBean allows for remote monitoring of log messages (see [Chapter 3, Logging \[417\]](#)). The MBean has a single notification that, if listened to, will add an appender to the root logger that will send log messages as notifications. This MBean currently only provides log messages when using the Log4J logging service. The name of the Log MBean is

```
kodo:type=log,name=LogMBean.
```

12.4.2. Kodo Pooling DataSource MBean

The Kodo DataSource (see [Section 4.1, “Using the Kodo DataSource” \[423\]](#)) is managed by an MBean. The name of the MBean is `kodo:type=PoolingDataSource`.

The Kodo DataSource has a number of attributes which allow for viewing / editing of pool settings, and viewing of pool statistics. The Kodo DataSource also has a number of statistic notifications.

12.4.3. Prepared Statement Cache MBean

The prepared statement cache (see `MaxCachedStatements` under [Section 4.1, “Using the Kodo DataSource” \[423\]](#)) is managed by an MBean. The name of the MBean is `kodo:type=PreparedStatementCache`.

The cache has a single `MaxCachedStatements` Attribute for viewing / editing the number of statements to cache, and a number of attributes for viewing cache statistics. The cache also has a number of statistic notifications.

12.4.4. Query Cache MBean

The query cache (see [Section 10.1.3, “Query Cache” \[566\]](#)) is managed by an MBean. The name of the MBean is `kodo:type=QueryCacheImpl`.

The cache has a `CacheSize` attribute for viewing / editing the size of the LRU cache, and a `SoftReferenceSize` attribute for editing the size of the soft cache. It also has a number of attributes for viewing cache statistics. The cache also has a `clear` operation and a number of statistic notifications.

12.4.5. Data Cache MBean

Each data cache (see [Section 10.1, “Data Cache” \[560\]](#)) is managed by an MBean. The name of the MBean is `kodo:type=DataCacheImpl,name=<cache name>` where `<cache name>` is the name of the cache.

Each cache has a number of attributes which allow for viewing / editing of cache settings, and viewing of cache statistics. Each cache also has a number of statistic notifications.

12.4.6. TimeWatch MBean

Each `TimeWatch` statistic is managed by an MBean. For information on creating `TimeWatch` statistics, see **TimeWatch** and **KodoTimeWatchManager** in the Kodo Javadoc. The name of the MBean is `kodo.TimeWatch:type=stat,name=<watchable name>,stat=<block name>` where `<watchable name>` is the name of the `TimeWatch` and where `<block name>` is the name of the named code block.

12.4.7. Runtime MBean

The runtime MBean allows for remote monitoring of the JVM in which Kodo is running. The MBean provides attributes for viewing the amount free memory available, and the total memory allocated to the JVM. It also has statistic notifications for free and total memory. The name of the runtime MBean is `kodo:type=runtime`.

12.4.8. Profiling MBean

The screenshot shows the SolarMetric JMX Viewer interface. The main window displays a tree view of the Profiling Agent MBean. The selected node is `Node.PersistenceManager`, which has a value of `2,415 ms`. The right-hand pane shows a table with columns for Field Name, Tot, Fetched (%), and Unfetched (%). The table contains two rows of data:

Field Name	Tot	Fetched (%)	Unfetched (%)
	422	0%	100%
	422	100%	0%

Below the table is a Statistics section with a table:

Name	value

The Kodo profiling MBean allows for profiling of application code. It is designed to help optimize the use of Kodo, and is not intended to be a generic profiling tool. Only Kodo-specific APIs are instrumented. This section describes how to setup the profiling capability within the Kodo Management Console. For more information about the profiling capability, see **Chapter 13, Profiling [603]**

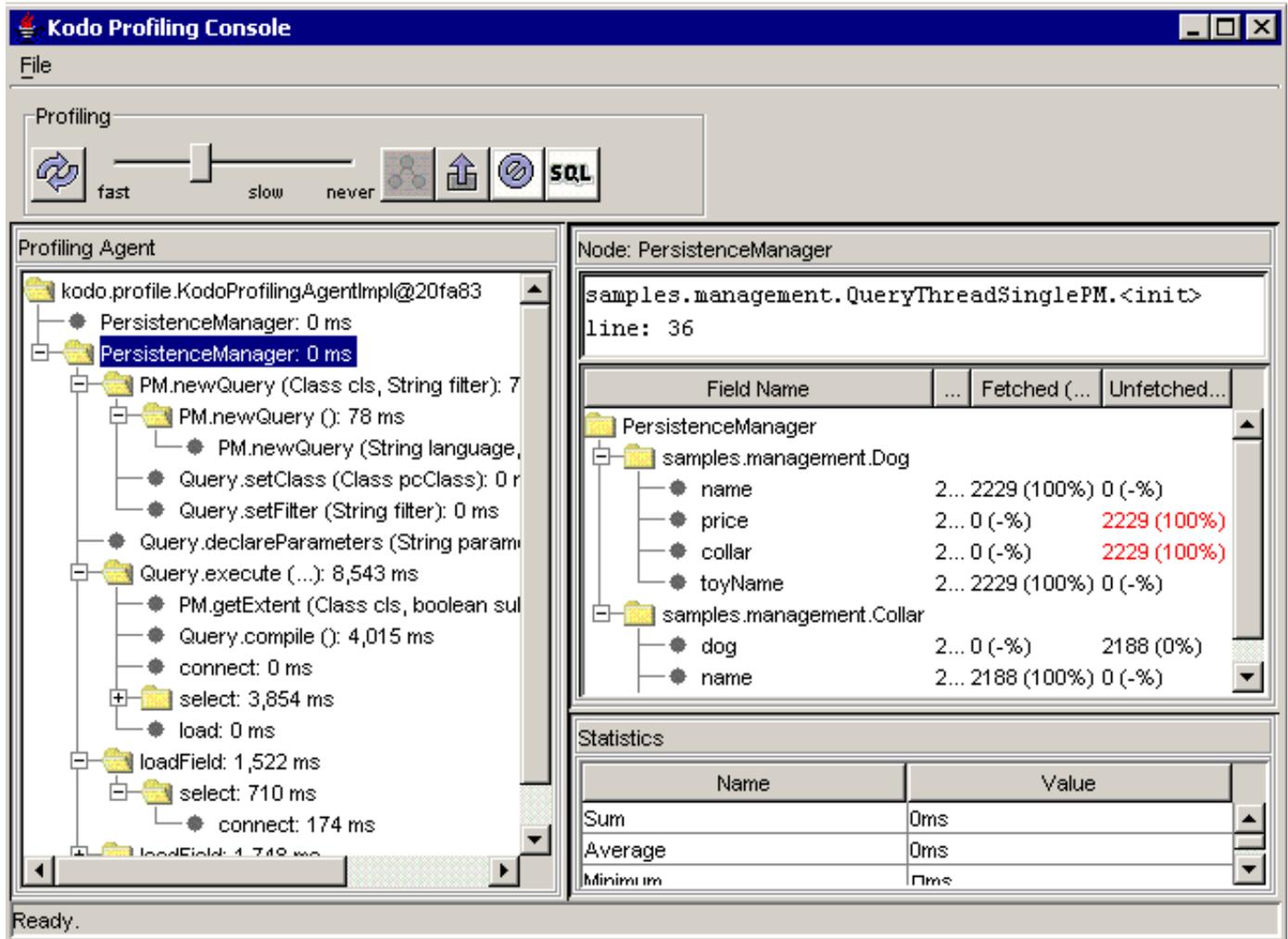
To use the profiling MBean, perform the following steps:

- Enable the MBean: Set the `kodo.ManagementConfiguration` property to `remote-mgmt-prof` or `local-mgmt-prof`.
- Start the Kodo Management Console either locally (see **Section 12.1, “Configuration” [591]**) or remotely (see **Section 12.2.1, “Remote Connection” [595]**).
- Select the MBean. The name of the MBean is `kodo:type=Profiling`.

- Listen to the MBean notifications. See **Section 12.2.2.1.2, “Listening to Notifications” [598]** (this happens automatically, but if you turn off notifications, you will need to turn them on again in order to see updates).

The MBean Panel contains a custom viewer when the profiling MBean is selected. This custom viewer contains the Kodo Profiling Console. Additionally, the console toolbar will have profiling toolbar options. Please see **Chapter 13, Profiling [603]** for more information.

Chapter 13. Profiling



The Kodo profiling capability allows for profiling of application code. It is designed to help optimize the use of Kodo, and is not intended to be a generic profiling tool. Only Kodo specific APIs are instrumented.

Important

This feature requires **Kodo Enterprise Edition** or **Kodo Standard Edition** with the **Performance Pack**.

The profiling capability can either be used standalone using the Kodo Profiling Console, or inside the Kodo Management Console using the Profiling MBean. To use the profiling capability within the Kodo Management Console see [Section 12.4.8, “Profiling MBean” \[601\]](#).

13.1. Profiling in an Embedded GUI

The profiling capability can be used standalone locally. To bring up the Kodo Profiling Console, set the following property (see [Section 2.7.41, “kodo.ManagementConfiguration” \[407\]](#)):

JPA XML format:

```
<property name="kodo.ManagementConfiguration" value="profiling-gui"/>
```

JDO properties format:

```
kodo.ManagementConfiguration: profiling-gui
```

The left pane of the Profiling Console contains a tree. Each node in the tree represents a call in a call stack. The root of each call stack is an `EntityManager` or `PersistenceManager`.

Each node may be composed of three items - the node name, the amount of time spent in each node, and the percentage of the total time spent in the parent node that this child node contributes. You can see details about a node in the right panes by selecting a node. The top right pane(s) yield detailed information about the node (e.g. the location in code where a transaction was started, or the SQL generated for a query), and the bottom right pane contains statistics about the node.

The Profiling Console has a Profiling Toolbar. It has the following controls:

- `Refresh`: Refreshes the statistics shown in the tree.
- `Refresh slider`: Set the interval at which the tree will be refreshed.
- `Show descendants`: Show the tree with `EntityManager` / `PersistenceManager` as roots.

This is the normal view. This option is available only when the `Show ancestors` view is in use. See the `Show ancestors` feature below.

- `Export . . .`: Export the current profiling data to a file for later viewing. Exported profiling data is stored in files ending in `.prx`. The export can be viewed using the `profilingviewer` application. To do this run the `profilingviewer` application and pass in the name of the exported file.
- `Reset`: Resets the statistics shown in the tree.
- `Show SQL`: Show an inverted tree where the root nodes are SQL statements, and their ancestors are shown as children in the tree.

The profiling call tree has a context menu containing the following options:

- `Refresh`: Refreshes the statistics shown in the tree.
- `Show ancestors`: Show an inverted tree where the nodes of the same name and description as the selected node are used as the roots, and their ancestors are shown as children in the tree.
- `Show descendants`: Show the tree with `EntityManager` / `PersistenceManager` as roots.

This is the normal view. This option is available only when the `Show ancestors` view is in use.

- `Export . . .`: Export the current profiling data to a file for later viewing. Exported profiling data is stored in files ending in `.prx`. The export can be viewed using the `profilingviewer` application. To do this run the `profilingviewer` application and pass in the name of the exported file.
- `Reset`: Resets the statistics shown in the tree.

The detail information for an `EntityManager` or `PersistenceManager` includes information about objects that were fetched in the context of that manager. The detail area consists of information about where in code the manager was created, and a table with information on each field in each persistence-capable class. The table has the following columns:

- `Field Name`: The name of the persistent field.
- `Total`: The total number of times the object containing that field was loaded in the context of the containing manager.
- `Fetches / % Used`: The first number represents the number of times the field was fetched during the initial load. The second number represents the percentage of initially fetched fields that are actually accessed. A low percentage indicates that perhaps that field should not be in the default fetch group or in a fetch group configured for initial load.
- `Unfetched / % Used`: The first number represents the number of times the field was *not* fetched during the initial load. The second number represents the percentage of the time the initially unfetched field is actually accessed. A high percentage indicates that perhaps that field should be in the default fetch group or in a fetch group configured for initial load.

13.2. Dumping Profiling Data to Disk from a Batch Process

The profiling capability can be used to create profiling exports. Exported profiling data is stored in files ending with the `.prx` suffix. To view exported profiling data, use the `profilingviewer` command, e.g. `profilingviewer myexport.prx`. To enable automatic export of profiling data, set the following properties (see [Section 2.7.41](#), “`kodo.ManagementConfiguration`” [407]):

JPA XML format:

```
<property name="kodo.ManagementConfiguration" value="profiling-export"/>
```

JDO properties format:

```
kodo.ManagementConfiguration: profiling-export
```

When exporting, the `ManagementConfiguration` value takes the following optional parameters:

- `IntervalMillis`: The number of milliseconds between exports (defaults to -1, indicating that there will be a single export upon exit).
- `Basename`: The basename of the exported data file to create.
- `UniqueNames`: A boolean that indicates whether or not the exported data file name should have the systems current time in milliseconds included as part of the name in order to make it unique.

For example, in order to export data every five minutes with a basename of `MyExport` set the `kodo.ManagementConfiguration` property to:

```
profiling-export(IntervalMillis=300000,Basename="MyExport")
```

13.3. Controlling How the Profiler Obtains Context Information

The description of `EntityManager / PersistenceManager` and transaction nodes are dependent on the call stack when the manager is created or the transaction is started, and can be controlled by an optional parameter to the `kodo.ManagementConfiguration` property (see [Section 2.7.41, “kodo.ManagementConfiguration” \[407\]](#)):

- `StackStyle`: Indicates how much of the call stack where the manager or transaction was created/started to include. `line` indicates a single line, `partial` indicates a partial stack starting at the user code, and `full` indicates a full call stack, including Kodo code. Defaults to `line`. The full call stack is only interesting from a debugging standpoint.

Chapter 14. Third Party Integration

Kodo provides a number of mechanisms for integrating with third-party tools. The following chapter will illustrate these integration features.

14.1. Apache Ant

Ant is a very popular tool for building Java projects. It is similar to the `make` command, but is Java-centric and has more modern features. Ant is open source, and can be downloaded from Apache's Ant web page at <http://jakarta.apache.org/ant/>. Ant has become the de-facto standard build tool for Java, and many commercial integrated development environments provide some support for using ant build files. The remainder of this section assumes familiarity with writing Ant `build.xml` files.

Kodo provides pre-built Ant task definitions for all bundled tools:

- **Enhancer Task**
- **Application Identity Tool Task**
- **Mapping Tool Task**
- **Reverse Mapping Tool Task**
- **Schema Tool Task**

The source code for all the ant tasks is provided with the distribution under the `src` directory. This allows you to customize various aspects of the ant tasks in order to better integrate into your development environment.

14.1.1. Common Ant Configuration Options

All Kodo tasks accept a nested `config` element, which defines the configuration environment in which the specified task will run. The attributes for the `config` tag are defined by the `JDBCConfiguration` bean methods. Note that excluding the `config` element will cause the Ant task to use the default system configuration mechanism, such as the configuration defined in the `kodo.xml` or `kodo.properties` file.

Following is an example of how to use the nested `config` tag in a `build.xml` file:

Example 14.1. Using the <config> Ant Tag

```
<mappingtool>
  <fileset dir="${basedir}">
    <include name="**/model/*.java" />
  </fileset>
  <config connectionUserName="scott" connectionPassword="tiger"
    connectionURL="jdbc:oracle:thin:@saturn:1521:solarsid"
    connectionDriverName="oracle.jdbc.driver.OracleDriver" />
</mappingtool>
```

It is also possible to specify a `properties` or `propertiesFile` attribute on the `config` tag, which will be used to locate a properties resource or file. The resource will be loaded relative to the current `CLASSPATH`.

Example 14.2. Using the Properties Attribute of the <config> Tag

```
<mappingtool>
  <fileset dir="${basedir}">
    <include name="**/model/*.java"/>
  </fileset>
  <config properties="kodo-dev.properties"/>
</mappingtool>
```

Example 14.3. Using the PropertiesFile Attribute of the <config> Tag

```
<mappingtool>
  <fileset dir="${basedir}">
    <include name="**/model/*.java"/>
  </fileset>
  <config propertiesFile="../conf/kodo-dev.properties"/>
</mappingtool>
```

Tasks also accept a nested `classpath` element, which you can use in place of the default classpath. The `classpath` argument behaves the same as it does for Ant's standard `javac` element. It is sometimes the case that projects are compiled to a separate directory than the source tree. If the target path for compiled classes is not included in the project's classpath, then a `classpath` element that includes the target class directory needs to be included so the enhancer and mapping tool can locate the relevant classes.

Following is an example of using a `classpath` tag:

Example 14.4. Using the <classpath> Ant Tag

```
<kodoc>
  <fileset dir="${basedir}/source">
    <include name="**/model/*.java" />
  </fileset>
  <classpath>
    <pathelement location="${basedir}/classes"/>
    <pathelement location="${basedir}/source"/>
    <pathelement path="${java.class.path}"/>
  </classpath>
</kodoc>
```

Finally, tasks that invoke code-generation tools like the application identity tool and reverse mapping tool accept a nested `codeformat` element. See the code formatting documentation in [Section 2.4.1, “Code Formatting” \[395\]](#) for a list of code formatting attributes.

Example 14.5. Using the <codeformat> Ant Tag

```
<reversemappingtool package="com.xyz.jdo" directory="${basedir}/src">
  <codeformat tabSpaces="4" spaceBeforeParen="true" braceOnSameLine="false"/>
</reversemappingtool>
```

Kodo's Ant tasks all utilize BEA licensing system and expect to find `license.bea` available in the classpath. Be sure that this file is available to every task. For further details, see [Section 2.2, “License Configuration” \[393\]](#).

14.1.2. Enhancer Ant Task

The enhancer task allows you to invoke the Kodo enhancer directly from within the Ant build process. The task's parameters correspond exactly to the long versions of the command-line arguments to `kodoc`.

Important

Set the tasks `jdoEnhance` attribute to `true` to perform JDO enhancement.

The enhancer task accepts a nested `fileset` tag to specify the files that should be processed. You can specify `.java`, `.jdo`, or `.class` files. If you do not specify any files, the task will run on the classes listed in your `kodo.MetaDataFactory` property.

Following is an example of using the enhancer task in a `build.xml` file:

Example 14.6. Invoking the Enhancer from Ant

```
<target name="enhance">
  <!-- define the kodoc task; this can be done at the top of the -->
  <!-- build.xml file, so it will be available for all targets -->
  <taskdef name="kodoc" classname="kodo.ant.PCEnhancerTask"/>

  <!-- invoke enhancer on all .jdo files below the current directory -->
  <kodoc jdoEnhance="true">
    <fileset dir=".">
      <include name="**/model/*.java" />
    </fileset>
  </kodoc>
</target>
```

14.1.3. Application Identity Tool Ant Task

The application identity tool task allows you to invoke the application identity tool directly from within the Ant build process. The task's parameters correspond exactly to the long versions of the command-line arguments to `appidtool`.

The application identity tool task accepts a nested `fileset` tag to specify the files that should be processed. You can specify `.java`, `.jdo`, or `.class` files. If you do not specify any files, the task will run on the classes listed in your `kodo.MetaDataFactory` property.

Following is an example of using the application identity tool task in a `build.xml` file:

Example 14.7. Invoking the Application Identity Tool from Ant

```

<target name="appids">
  <!-- define the appidtool task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="appidtool" classname="kodo.ant.ApplicationIdToolTask"/>

  <!-- invoke tool on all .jdo files below the current directory -->
  <appidtool>
    <fileset dir=".">
      <include name="**/model/*.java" />
    </fileset>
    <codeformat spaceBeforeParen="true" braceOnSameLine="false"/>
  </appidtool>
</target>

```

14.1.4. Mapping Tool Ant Task

The mapping tool task allows you to directly invoke the mapping tool from within the Ant build process. It is useful for making sure that the database schema and object-relational mapping data is always synchronized with your persistent class definitions, without needing to remember to invoke the mapping tool manually. The task's parameters correspond exactly to the long versions of the command-line arguments to the `mappingtool`.

The mapping tool task accepts a nested `fileset` tag to specify the files that should be processed. You can specify `.java`, `.jdo`, or `.class` files. If you do not specify any files, the task will run on the classes listed in your `kodo.MetaDataFactory` property.

Following is an example of a `build.xml` target that invokes the mapping tool:

Example 14.8. Invoking the Mapping Tool from Ant

```

<target name="refresh">
  <!-- define the mappingtool task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="mappingtool" classname="kodo.jdbc.ant.MappingToolTask"/>

  <!-- add the schema components for all .jdo files below the -->
  <!-- current directory -->
  <mappingtool action="buildSchema">
    <fileset dir=".">
      <include name="**/*.jdo" />
    </fileset>
  </mappingtool>
</target>

```

14.1.5. Reverse Mapping Tool Ant Task

The reverse mapping tool task allows you to directly invoke the reverse mapping tool from within Ant. While many users will only run the reverse mapping process once, others will make it part of their build process. The task's parameters correspond exactly to the long versions of the command-line arguments to the `reversemappingtool`.

Following is an example of a `build.xml` target that invokes the reverse mapping tool:

Example 14.9. Invoking the Reverse Mapping Tool from Ant

```

<target name="reversemap">

```

```
<!-- define the reversmappingtool task; this can be done at the top of -->
<!-- the build.xml file, so it will be available for all targets -->
<taskdef name="reversmappingtool"
  classname="kodo.jdbc.ant.ReverseMappingToolTask"/>

<!-- reverse map the entire database -->
<reversmappingtool package="com.xyz.model" directory="{basedir}/src"
  customizerProperties="{basedir}/conf/reverse.properties">
  <codeformat tabSpaces="4" spaceBeforeParen="true" braceOnSameLine="false"/>
</reversmappingtool>
</target>
```

14.1.6. Schema Tool Ant Task

The schema tool task allows you to directly invoke the schema tool from within the Ant build process. The task's parameters correspond exactly to the long versions of the command-line arguments to the **schematool**.

Following is an example of a `build.xml` target that invokes the schema tool:

Example 14.10. Invoking the Schema Tool from Ant

```
<target name="schema">
  <!-- define the schematool task; this can be done at the top of -->
  <!-- the build.xml file, so it will be available for all targets -->
  <taskdef name="schematool" classname="kodo.jdbc.ant.SchemaToolTask"/>

  <!-- add the schema components for all .schema files below the -->
  <!-- current directory -->
  <schematool action="add">
    <fileset dir=".">
      <include name="**/*.schema" />
    </fileset>
  </schematool>
</target>
```

14.2. Maven

Maven plugins for Kodo are available from <http://maven-plugins.sourceforge.net/maven-kodo-plugin/>. These plugins are not developed by BEA. Please consult Maven plugins site above for support and licensing questions.

Chapter 15. Optimization Guidelines

There are numerous techniques you can use in order to ensure that Kodo operates in the fastest and most efficient manner. Following are some guidelines. Each describes what impact it will have on performance and scalability. Note that general guidelines regarding performance or scalability issues are just that - guidelines. Depending on the particular characteristics of your application, the optimal settings may be considerably different than what is outlined below.

In the following table, each row is labeled with a list of italicized keywords. These keywords identify what characteristics the row in question may improve upon. Many of the rows are marked with one or both of the *performance* and *scalability* labels. It is important to bear in mind the differences between performance and scalability (for the most part, we are referring to system-wide scalability, and not necessarily only scalability within a single JVM). The performance-related hints will probably improve the performance of your application for a given user load, whereas the scalability-related hints will probably increase the total number of users that your application can service. Sometimes, increasing performance will decrease scalability, and vice versa. Typically, options that reduce the amount of work done on the database server will improve scalability, whereas those that push more work onto the server will have a negative impact on scalability.

Table 15.1. Optimization Guidelines

Optimize database indexes <i>performance, scalability</i>	<p>The default set of indexes created by Kodo's mapping tool may not always be the most appropriate for your application. Manually setting indexes in your mapping metadata or manually manipulating database indexes to include frequently-queried fields (as well as dropping indexes on rarely-queried fields) can yield significant performance benefits.</p> <p>A database must do extra work on insert, update, and delete to maintain an index. This extra work will benefit selects with WHERE clauses, which will execute much faster when the terms in the WHERE clause are appropriately indexed. So, for a read-mostly application, appropriate indexing will slow down updates (which are rare) but greatly accelerate reads. This means that the system as a whole will be faster, and also that the database will experience less load, meaning that the system will be more scalable.</p> <p>Bear in mind that over-indexing is a bad thing, both for scalability and performance, especially for applications that perform lots of inserts, updates, or deletes.</p>
Use the best JDBC driver <i>performance, scalability, reliability</i>	<p>The JDBC driver provided by the database vendor is not always the fastest and most efficient. Some JDBC drivers do not support features like batched statements, the lack of which can significantly slow down Kodo's data access and increase load on the database, reducing system performance and scalability.</p>
JVM optimizations <i>performance, reliability</i>	<p>Manipulating various parameters of the Java Virtual Machine (such as hotspot compilation modes and the maximum memory) can result in performance improvements. For more details about optimizing the JVM execution environment, please see http://java.sun.com/docs/hotspot/PerformanceFAQ.html.</p>
Use the data cache <i>performance, scalability</i>	<p>Using Kodo's data and query caching features can often result in a dramatic improvement in performance. Additionally, these caches can significantly reduce the amount of load on the database, increasing the scalability characteristics of your application. Also, be sure to read about the concurrent cache option to see if it fits your needs.</p>
Set LargeTransaction to true, or set PopulateDataCache to false <i>performance vs. scalability</i>	<p>When using Kodo's data caching features (available in Kodo JDO Performance Pack and Enterprise Edition) in a transaction that will delete, modify, or create a very large number of objects you can set <code>LargeTransaction</code> to true and perform periodic flushes during your transaction to reduce its memory requirements. See the Javadoc: KodoEntityManager.setLargeTransaction, KodoPersistenceManager.setLargeTransaction Note that transactions in large mode have to more aggressively flush items from the data cache.</p> <p>If your transaction will visit objects that you know are very unlikely to be accessed by other transactions, for example an exhaustive report run only once a month, you can turn off population of the data cache so that the transaction doesn't fill the entire data cache with objects that won't be accessed again.</p>

	Again, see the Javadoc: KodoEntityManager.setPopulateDataCache , KodoPersistenceManager.setPopulateDataCache
Disable logging, performance tracking <i>performance</i>	Developer options such as verbose logging and the JDBC performance tracker can result in serious performance hits for your application. Before evaluating Kodo's performance, these options should all be disabled.
Use the Kodo Profiler <i>performance</i>	Take advantage of the Kodo Profiler described in Chapter 13, Profiling [603] to discover where your application is spending the most time, and to recognize misconfigured fetch groups.
Set IgnoreChanges to true, or set FlushBeforeQueries to true <i>performance vs. scalability</i>	<p>When both the <code>kodo.IgnoreChanges</code> and <code>kodo.FlushBeforeQueries</code> properties are set to false, Kodo needs to consider in-memory dirty instances during queries. This can sometimes result in Kodo needing to evaluate the entire extent objects in order to return the correct query results, which can have drastic performance consequences. If it is appropriate for your application, configuring <code>FlushBeforeQueries</code> to automatically flush before queries involving dirty objects will ensure that this never happens. Setting <code>IgnoreChanges</code> to false will result in a small performance hit even if <code>FlushBeforeQueries</code> is true, as incremental flushing is not as efficient overall as delaying all flushing to a single operation during commit. This is because incrementally flushing decreases Kodo's ability to maximize statement batching, and increases resource utilization.</p> <p>Note that the default setting of <code>FlushBeforeQueries</code> is <code>with-connection</code>, which means that data will be flushed only if a dedicated connection is already in use by the <code>EntityManager</code> or <code>PersistenceManager</code>. So, the default value may not be appropriate for you.</p> <p>Setting <code>IgnoreChanges</code> to true will help performance, since dirty objects can be ignored for queries, meaning that incremental flushing or client-side processing is not necessary. It will also improve scalability, since overall database server usage is diminished. On the other hand, setting <code>IgnoreChanges</code> to false will have a negative impact on scalability, even when using automatic flushing before queries, since more operations will be performed on the database server.</p>
Configure kodo.ConnectionRetainMode appropriately <i>performance vs. scalability</i>	<p>The <code>ConnectionRetainMode</code> configuration option controls when Kodo will obtain a connection, and how long it will hold that connection. The optimal settings for this option will vary considerably depending on the particular behavior of your application. You may even benefit from using different retain modes for different parts of your application.</p> <p>The default setting of <code>on-demand</code> minimizes the amount of time that Kodo holds onto a datastore connection. This is generally the best option from a scalability standpoint, as database resources are held for a minimal amount of time. However, if your connection pool is overly small relative to the number of concurrent sessions that need access to the database, or if your <code>DataSource</code> is not efficient at managing its pool, then this default value could cause undesirable pool contention.</p>
Ensure that batch updates are available <i>performance, scalability</i>	When performing bulk inserts, updates, or deletes, Kodo will use batched statements. If this feature is not available in your JDBC driver, then Kodo will need to issue multiple SQL statements instead of a single batch statement.
Use flat inheritance <i>performance, scalability vs. disk space</i>	<p>Mapping inheritance hierarchies to a single database table is faster for most operations than other strategies employing multiple tables. If it is appropriate for your application, you should use this strategy whenever possible.</p> <p>However, this strategy will require more disk space on the database side. Disk space is relatively inexpensive, but if your object model is particularly large, it can become a factor.</p>
High sequence increment <i>performance, scalability</i>	For applications that perform large bulk inserts, the retrieval of sequence numbers can be a bottleneck. Increasing sequence increments and using table-based rather than native database sequences can reduce or eliminate this bottleneck. In some cases, implementing your own sequence factory can further optimize sequence number retrieval.
Use optimistic transactions <i>performance, scalability</i>	Using datastore transactions translates into pessimistic database row locking, which can be a performance hit (depending on the database). If appropriate for your application, optimistic transactions are typically faster than datastore transactions.

	<p>Optimistic transactions provide the same transactional guarantees as datastore transactions, except that you must handle a potential optimistic verification exception at the end of a transaction instead of assuming that a transaction will successfully complete. In many applications, it is unlikely that different concurrent transactions will operate on the same set of data at the same time, so optimistic verification increases the concurrency, and therefore both the performance and scalability characteristics, of the application. A common approach to handling optimistic verification exceptions is to simply present the end user with the fact that concurrent modifications happened, and require that the user redo any work.</p>
<p>Use query aggregates and projections <i>performance, scalability</i></p>	<p>Using aggregates to compute reporting data on the database server can drastically speed up queries. Similarly, using projections when you are interested in specific object fields or relations rather than the entire object state can reduce the amount of data Kodo must transfer from the database to your application.</p>
<p>Always close resources <i>scalability</i></p>	<p>Under certain settings, <code>EntityManagers</code>, <code>PersistenceManagers</code>, <code>Extent</code> iterators, and <code>Query</code> results may be backed by resources in the database.</p> <p>For example, if you have configured Kodo to use scrollable cursors and lazy object instantiation by default, each query result will hold open a <code>ResultSet</code> object, which, in turn, will hold open a <code>Statement</code> object (preventing it from being re-used). Garbage collection will clean up these resources, so it is never necessary to explicitly close them, but it is always faster if it is done at the application level.</p>
<p>Optimize connection pool settings <i>performance, scalability</i></p>	<p>Kodo's built-in connection pool's default settings may not be optimal for all applications. For applications that instantiate and close many <code>EntityManagers</code> or <code>PersistenceManagers</code> (such as a web application), increasing the size of the connection pool will reduce the overhead of waiting on free connections or opening new connections.</p> <p>You may want to tune the prepared statement pool size with the connection pool size.</p>
<p>Use detached state managers <i>performance</i></p>	<p>Attaching and even persisting instances can be more efficient when your detached objects use detached state managers. By default, Kodo does not use detached state managers when serializing an instance across tiers. See Section 11.1.3, “Defining the Detached Object Graph” [578] for how to force Kodo to use detached state managers across tiers, and for other options for more efficient attachment.</p> <p>The downside of using a detached state manager across tiers is that your enhanced persistent classes and the Kodo libraries must be available on the client tier.</p>
<p>Utilize the Entity-Manager and PersistenceManager caches <i>performance, scalability</i></p>	<p>When possible and appropriate, re-using <code>EntityManagers</code> and <code>PersistenceManagers</code> and setting the <code>RetainState</code> configuration option to <code>true</code> may result in significant performance gains, since the manager's built-in object cache will be used.</p>
<p>Enable multithreaded operation only when necessary <i>performance</i></p>	<p>Kodo respects the <code>kodo.Multithreaded</code> option in that it does not impose synchronization overhead for applications that set this value to <code>false</code>. If your application is guaranteed to only use single-threaded access to Kodo resources and persistent objects, setting this option to <code>false</code> will result in the elimination of synchronization overhead, and may result in a modest performance increase.</p>
<p>Enable large data set handling <i>performance, scalability</i></p>	<p>If you execute queries that return large numbers of objects or have relations (collections or maps) that are large, and if you often only access parts of these data sets, enabling large result set handling where appropriate can dramatically speed up your application, since Kodo will bring the data sets into memory from the database only as necessary.</p>
<p>Disable large data set handling <i>performance, scalability</i></p>	<p>If you have enabled scrollable result sets and on-demand loading but do not require it, consider disabling it again. Some JDBC drivers and databases (SQLServer for example) are much slower when used with scrolling result sets.</p>

<p>Use short discriminator values, or turn off the discriminator</p> <p><i>performance, scalability</i></p>	<p>The default discriminator strategy of storing the class name in the discriminator column is quite robust, in that it can handle any class and needs no configuration, but the downside of this robustness is that it puts a relatively lengthy string into each row of the database. With a little application-specific configuration, you can easily reduce this to a single character or integer. This can result in significant performance gains when dealing with many small objects, since the subclass indicator data can become a significant proportion of the data transferred between the JVM and the database.</p> <p>Alternately, if certain persistent classes in your application do not make use of inheritance, then you can disable the discriminator for these classes altogether.</p>
<p>Use the DynamicSchemaFactory</p> <p><i>performance, validation</i></p>	<p>If you are using a <code>kodo.jdbc.SchemaFactory</code> setting of something other than the default of <code>dynamic</code>, consider switching back. While other factories can ensure that object-relational mapping information is valid when a persistent class is first used, this can be a slow process. Though the validation is only performed once for each class, switching back to the <code>DynamicSchemaFactory</code> can reduce the warm-up time for your application.</p>
<p>Do not use XA transactions</p> <p><i>performance, scalability</i></p>	<p>XA transactions can be orders of magnitude slower than standard transactions. Unless distributed transaction functionality is required by your application, use standard transactions.</p> <p>Recall that XA transactions are distinct from managed transactions - managed transaction services such as that provided by EJB declarative transactions can be used both with XA and non-XA transactions. XA transactions should only be used when a given business transaction involves multiple different transactional resources (an Oracle database and an IBM transactional message queue, for example).</p>
<p>Use Sets instead of List/Collections</p> <p><i>performance, scalability</i></p>	<p>There is a small amount of extra overhead for Kodo to maintain collections where each element is not guaranteed to be unique. If your application does not require duplicates for a collection, you should always declare your fields to be of type <code>Set</code>, <code>SortedSet</code>, <code>HashSet</code>, or <code>TreeSet</code>.</p>
<p>Use query parameters instead of encoding search data in filter strings</p> <p><i>performance</i></p>	<p>If your queries depend on parameter data only known at runtime, you should use query parameters rather than dynamically building different query strings. Kodo performs aggressive caching of query compilation data, and the effectiveness of this cache is diminished if multiple query filters are used where a single one could have sufficed.</p>
<p>Tune your fetch groups appropriately</p> <p><i>performance, scalability</i></p>	<p>The fetch groups used when loading an object control how much data is eagerly loaded, and by extension, which fields must be lazily loaded at a future time. The ideal fetch group configuration loads all the data that is needed in one fetch, and no extra fields - this minimizes both the amount of data transferred from the database, and the number of trips to the database.</p> <p>If extra fields are specified in the fetch groups (in particular, large fields such as binary data, or relations to other persistence-capable objects), then network overhead (for the extra data) and database processing (for any necessary additional joins) will hurt your application's performance. If too few fields are specified in the fetch groups, then Kodo will have to make additional trips to the database to load additional fields as necessary.</p>
<p>Use eager fetching</p> <p><i>performance, scalability</i></p>	<p>Using eager fetching when loading subclass data or traversing relations for each instance in a large collection of results can speed up data loading by orders of magnitude.</p>

Part 7. Kodo JPA/JDO Samples

Table of Contents

1. Kodo Sample Code	618
1.1. JDO - JPA Persistence Interoperability	618
1.2. JPA	619
1.2.1. Sample Human Resources Model	619
1.2.2. JMX Management	619
1.3. JDO	620
1.3.1. Using Application Identity	620
1.3.2. Customizing Logging	620
1.3.3. Custom Sequence Factory	620
1.3.4. Using Externalization to Persist Second Class Objects	620
1.3.5. Custom Mappings	621
1.3.6. Custom Proxies	621
1.3.7. Example of full-text searching in JDO	621
1.3.8. Horizontal Mappings	622
1.3.9. Sample Human Resources Model	622
1.3.10. Sample School Schedule Model	622
1.3.11. JMX Management	623
1.3.12. XML Store Manager	624
1.3.13. Using JDO with Java Server Pages (jsp)	625
1.3.14. JDO Enterprise Java Beans 2.x Facade	625

Chapter 1. Kodo Sample Code

The Kodo distribution comes with a number of examples that illustrate the usage of various features.

1.1. JDO - JPA Persistence Interoperability

This sample demonstrates how to combine JDO and JPA in a single application. The `MachineMain.java` program uses both `EntityManagers` and `PersistenceManagers` in a single transaction including persist, delete and query operations.

The sample includes both annotated persistent classes as well as JDOR metadata information. The application can switch to either system simply by changing the bootstrap mechanism. Depending on which configuration system you use, Kodo will read the corresponding metadata format. You can override some or all of this behavior using Kodo's configuration options, such as **`kodo.MetadataFactory`**.

To use this sample, you should ensure that either a `jdo.properties` or `persistence.xml` are in the `META-INF` directory in your `CLASSPATH`. The rest of the files for this sample are located in the `samples/mixed` directory of the Kodo installation. This tutorial requires JDK 5. To run this tutorial:

- Ensure that your environment is set properly as described in the README and that your current path is in the mixed sample directory.
- Edit the configuration file you are using to include your license key. You may want to edit `ConnectionURL` to point to an absolute URL (e.g. `C:/kodo/mixed-sample-db`) if using a file-based database like HSQL.
- Include the list of persistent classes in your configuration file. For JPA, you will want to add the following lines to `persistence.xml` before the `<property>` lines:

```
<class>samples.mixed.Machine</class>
<class>samples.mixed.Crane</class>
<class>samples.mixed.Bulldozer</class>
<class>samples.mixed.Operator</class>
```

If you are using JDO, point the metadata factory at the `.jdo` resource containing your persistent classes:

```
kodo.MetadataFactory: Resources=samples/mixed/package.jdo
```

- Compile the classes:

```
javac *.java
```

- You should then proceed to pass in the configuration file you are using to the enhancer:

```
kodoc -p persistence.xml Machine.java Crane.java Bulldozer.java Operator.java
```

or

```
jdoc -p jdo.properties Machine.java Crane.java Bulldozer.java Operator.java
```

- Similarly, you should pass in the same argument to `mappingtool`:

```
mappingtool -p persistence.xml -a buildSchema Machine.java Crane.java Bulldozer.java
```

```
Operator.java
```

or

```
mappingtool -p jdo.properties -a buildSchema Machine.java Crane.java Bulldozer.java  
Operator.java
```

- You can now run the sample application. The first argument is which operation you want the program to run. The second argument tells the application which bootstrap system to use:

```
java samples.mixed.MachineMain <create | delete> <jdo | jpa>
```

1.2. JPA

1.2.1. Sample Human Resources Model

The files for this sample are located in the `samples/persistence/models/humres` directory of the Kodo installation. This sample demonstrates the mapping of an example "Human Resources" schema. The following concepts are illustrated in this sample:

- Value Mappings
- One to One Mappings
- One to Many Mappings (with and without inverses)

1.2.2. JMX Management

This sample shows how to use the Kodo's JMX Management features. In order to see an example of runtime management:

- Ensure that the Kodo distribution root directory is in your CLASSPATH
- Ensure that an appropriate `META-INF/persistence.xml` is in a directory in your CLASSPATH or in the root level of a jar in your CLASSPATH
- Ensure that your CLASSPATH has all of the jars distributed with Kodo
- `javac *.java`
- `kodoc -p persistence.xml Animal.java Dog.java Collar.java`
- `mappingtool -p persistence.xml Animal.java Dog.java Collar.java`
- Enumerate the persistent classes in your `persistence.xml` file:

```
<class>samples.persistence.management.Animal</class> <property  
name="samples.persistence.management.Dog> <property  
name="samples.persistence.management.Collar>
```
- `java samples.persistence.management.SeedDatabase`
- Add the following line to your `persistence.xml` file: `<property name="kodo.ManagementConfiguration" value="local-mgmt"/>`

- Run the sample program which will bring up Kodo's management console upon initialization: `java samples.persistence.management.ManagementMain`

The Kodo ProfilingAgent also has a JMX MBean. You can see an example of profiling if you turn on the ProfilingAgent by making the following setting in your `persistence.xml` file:

- `<property name="kodo.ManagementConfiguration" value="local-mgmt-prof"/>`

The `Dog` class includes a field `collar` which has been marked as a lazily loaded field. After seeing the profiling data, you may want to enable eager fetching on the field to see how the performance profile changes. Edit `Dog.java` and change the `@OneToOne` annotation fetch attribute to `FetchType.EAGER`. Recompile and reenhance `Dog` and run the sample again to see the changes.

Note that the Kodo data cache also has a JMX MBean. You can see an example of cache management if you turn on the data cache by adding the following settings to your `persistence.xml` file:

- `<property name="kodo.DataCache" value="true"/>`
- `<property name="kodo.RemoteCommitProvider" value="sjvm"/>`

Kodo's management and profiling tools can be used in a wide range of modes and environments. For example, by setting the `kodo.ManagementConfiguration` setting to `profiling-gui`, you can use the profiling tool exclusively. You can have Kodo connect and integrate with your existing JMX configuration such as provided by your application server or products such as MX4J. For further details on configuring Kodo management and profiling tools, see [Chapter 12, *Management and Monitoring* \[591\]](#).

To extend Kodo's monitoring abilities to your own code, you can use `TimeWatch`. An example of how to use the `TimeWatch` can be found in `QueryThread.java`. A `TimeWatch` allows for monitoring of named code blocks.

1.3. JDO

1.3.1. Using Application Identity

The files for this sample are located in the `samples/jdo/appid` directory of the Kodo installation. This sample shows how to use JDO application identity. The `GovernmentForm` class uses a static inner class as its application identity class. The `Main` class is a simple JDO application that allows you to create and manipulate persistent `GovernmentForm` instances.

1.3.2. Customizing Logging

The files for this sample are located in the `samples/jdo/logging` directory of the Kodo installation. This sample shows how to plug a custom logging strategy into Kodo. The `Main` class implements custom logging and has a `main` method demonstrating how to set your custom log as the system default.

1.3.3. Custom Sequence Factory

The files for this sample are located in the `samples/jdo/seqfactory` directory of the Kodo installation. This sample shows how to define a custom JDO sequence factory and use it for certain classes.

1.3.4. Using Externalization to Persist Second Class Objects

The files for this sample are located in the `samples/jdo/externalization` directory of the Kodo installation. This sample demonstrates how to persist field types that aren't directly supported by JDO using Kodo's externalization framework.

The `ExternalizationFields` class is a persistence-capable class with fields of various types, none of which are recognized by JDO. The JDO metadata for `ExternalizationFields` uses Kodo's "externalizer" and "factory" metadata extensions to name methods that can be used to transform each field into a supported type, and then reconstruct it from its external form. Even complex external forms are supported; the `ExternalizationFields.pair` field externalizes to a list of persistence-capable objects.

The `ExternalizationFieldsMain` class is a driver to demonstrate that `ExternalizationFields` instances persist correctly.

1.3.5. Custom Mappings

The files for this sample are located in the `samples/jdo/ormapping` directory of the Kodo installation. This sample demonstrates custom field and class mapping.

- `IsMaleHandler` is a custom value handler that transforms a boolean field into 'M' or 'F' characters in the database. It shows how to create transformation value handlers.
- `PointHandler` is a custom value handler for `java.awt.Point`. It demonstrates how to create a multi-column custom value handlers for complex data.
- `XMLStrategy` is a custom field strategy that simulates a field that maps to a non-standard column type, and that may require non-standard operations to store and retrieve data.
- `StoredProcClassStrategy` is a custom class strategy that simulates using stored procedures to access persistent data.

The `CustomFields` class is a persistence-capable class that uses each of the custom field mappings above. The `StoredProc` class is a persistence-capable class that uses the `StoredProcClassStrategy`. The `CustomFieldsMain` and `StoredProcMain` classes are simple driver programs for each of these types.

Note that creating custom class mappings requires a license for the Enterprise Edition. Custom field and value mappings work with the Standard Edition.

Also, make sure to browse the "externalization" sample directory. Kodo includes an externalization feature that can be used to persist many unsupported field types without having to create a custom mapping.

1.3.6. Custom Proxies

The files for this sample are located in the `samples/jdo/proxies` directory of the Kodo installation. This sample demonstrates custom second class object proxies for mutable persistent fields not directly supported by Kodo.

- `CustomSet` is a custom extension of `java.util.Set` with some extra methods.
- `CustomStringContainer` is a mutable object that holds an internal string. We use this object as a second class object, not a first class object.

The `CustomProxies` class is a persistence-capable class that uses each of the custom proxies above. `CustomProxiesMain` is a simple driver program for showing that the custom proxies work.

1.3.7. Example of full-text searching in JDO

The files for this sample are located in the `samples/jdo/textindex` directory of the Kodo installation. This sample demon-

strates how full-text indexing might be implemented in JDO. Most relational databases cannot optimize contains queries for large text fields, meaning that any substring query will result in a full table scan (which can be extremely slow for tables with many rows).

The `AbstractIndexable` class implements `javax.jdo.InstanceCallbacks` which will cause the textual content of the implementing persistent class to be split into individual "word" tokens, and stored in a related table. Since this happens whenever an instance of the class is stored, the index is always up to date. The `Indexer` class is a utility that assists in building queries that act on the indexed field.

The `TextIndexMain` class is a driver to demonstrate a simple text indexing application.

1.3.8. Horizontal Mappings

This sample demonstrates how to use horizontal mappings in JDO to implement the common pattern of individual tables that track the creation data and last modification date. This is accomplished through the `LastModified` superclass that contains a `lastModificationDate` and `creationDate` field that is automatically updated from the `jdoPreStore` method.

The `package.jdo` file specifies that the `LastModified` class use a horizontal mapping, which means that each of the fields will then be declared in the subclasses: `Widget`, `WidgetOrder`, `WidgetOrderItem`.

1.3.9. Sample Human Resources Model

The files for this sample are located in the `samples/jdo/models/humres` directory of the Kodo installation. This sample demonstrates the JDO mapping of an example "Human Resources" schema. The following concepts are illustrated in this sample:

- Mixed Application Identity and Datastore Identity
- Named Query execution
- Value Mappings
- One to One Mappings
- One to Many Mappings (with and without inverses)

1.3.10. Sample School Schedule Model

The files for this sample are located in the `samples/jdo/models/school` directory of the Kodo installation. This sample demonstrates different JDO inheritance strategies for the same object model. The following concepts are illustrated in this sample:

- Flat Inheritance Mappings
- State Image Version Indicator
- Vertical Inheritance Mappings
- Horizontal Inheritance Mappings
- Value Mappings
- Many to Many Mappings

The `samples/models/school` directory contains the following interfaces that represent the abstract data model:

- Address
- HomeAddress (extends Address)
- WorkAddress (extends Address)
- Person
- Parent (extends Person)
- Student (extends Person)
- Employee (extends Person)
- Teacher (extends Employee)
- Admin (extends Employee)
- Staff (extends Employee)
- Course
- ScheduledClass

Each of the `flat`, `vertical`, and `horizontal` subdirectories contain implementations of the object model that are identical except for name, and that their inheritance mapping in the `package.jdo` file uses the type of the directory's corresponding name.

1.3.11. JMX Management

This sample shows how to use the Kodo JMX Management features. In order to see an example of runtime management:

In order to see an example of remote management:

- Open two console windows, and ensure in each that your `CLASSPATH` has all of the jars distributed with Kodo, and that your execution path is configured for the Kodo tools. If you need additional help with this step, see the installation directions at **Chapter 2, *Kodo Installation* [4]**.
- In the first console window, ensure that the Kodo distribution root directory is in your `CLASSPATH`
- Ensure that an appropriate `jdo.properties` is in a directory in your `CLASSPATH` or in the root level of a jar in your `CLASSPATH`, or in a `META-INF` directory in your `CLASSPATH`.
- Change to the `samples/management` directory
- Compile the class file by running: **`javac *.java`**
- Enhance the data classes by running: **`jdoc package.jdo`**
- Update the database schema and generate mappings by running: **`mappingtool package.jdo`**
- Seed the database by running: **`java samples.management.SeedDatabase`**
- Add the following line to your properties file:
`kodo.ManagementConfiguration: local-mgmt`
- Start the sample program which will bring up Kodo's management console upon initialization. **`java`**

samples.management.ManagementSampleMain

The Kodo ProfilingAgent also has a JMX MBean. You can see an example of remote profiling if you turn on the ProfilingAgent by making the following setting in your properties file:

- **kodo.ManagementConfiguration: local-mgmt-prof**

A fetch group `g` has been pre-configured in the `package.jdo` file. You can turn on the usage of that fetch group in the example by adding the following setting to your properties file:

- **kodo.FetchGroups: g**

Note that the Kodo PersistenceManagerFactory DataCache also has a JMX MBean. You can see an example of DataCache management if you turn on the data cache by adding the following settings to your properties file:

- **kodo.DataCache: true**
- **kodo.RemoteCommitProvider: sjvm**

Kodo's management and profiling tools can be used in a wide range of modes and environments. For example, by setting the `kodo.ManagementConfiguration` setting to `profiling-gui`, you can use the profiling tool exclusively. You can have Kodo connect and integrate with your existing JMX configuration such as provided by your application server or products such as MX4J. For further details on configuring Kodo management and profiling tools, see **Chapter 12, Management and Monitoring [591]**.

To extend Kodo's monitoring abilities to your own code, you can use `TimeWatch`. An example of how to use the `TimeWatch` can be found in `QueryThread.java`. A `TimeWatch` allows for monitoring of named code blocks.

1.3.12. XML Store Manager

This sample shows how to use the Kodo XML Store Manager `AbstractStoreManager` implementation.

For more information on implementing your own custom store manager, please see the XML Store Manager Javadoc (package `kodo.xmlstore`), and the source code in `src/kodo/xmlstore` under your Kodo installation.

Note

The `AbstractStoreManager` requires a Kodo Enterprise Edition license.

In order to see an example of the XML Store Manager:

- Ensure that the Kodo distribution root directory is in your `CLASSPATH`. This is done automatically if you set up your environment as described in the README.
- Ensure that the sample `xmlstore.properties` is in a directory in your `CLASSPATH` or in the root level of a jar in your `CLASSPATH`. Enter your Kodo license key into the sample `xmlstore.properties` as the value of the **kodo.LicenseKey** property.
- Ensure that your `CLASSPATH` has all of the jars distributed with Kodo. This is done automatically if you set up your environment as described in the README.

- `javac *.java`
- `jdoc -p xmlstore.properties package.jdo`
- Note that there is no need to run `mappingtool` for custom store managers.
- `java samples.jdo.xmlstore.SeedDatabase`
- Note the XML data in the selected data directory as specified by your `kodo.ConnectionURL`.

1.3.13. Using JDO with Java Server Pages (jsp)

The files for this sample are located in the `samples/jdo/jsp` directory of the Kodo installation. This sample creates a simple Petshop web application to demonstrate using JDO with JSPs. Run `ant` on the included `build.xml` file and follow the instructions `ant` displays.

1.3.14. JDO Enterprise Java Beans 2.x Facade

The files for this sample are located in the `samples/jdo/ejb` directory of the Kodo installation. This sample demonstrates how to use JDO with 2.x EJBs. It includes session beans that use JDO instead of EJB 2.x entity beans.

Appendix 1. JPA Resources

- **EJB 3 JSR page**
- **Kodo JPA community support groups**
- **Sun EJB page**
- **Locally mirrored javax.persistence Javadoc**
- **Kodo API Javadoc**
- **Full Kodo Javadoc**
- **Locally mirrored JPA specification**

Appendix 2. JDO Resources

- **JDO JSR page**
- **Kodo community support groups**
- **Sun JDO page**
- **Locally mirrored javax.jdo Javadoc**
- **Kodo API Javadoc**
- **Full Kodo Javadoc**
- **Locally mirrored JDO public review draft**

Appendix 3. Kodo Editions

Kodo JPA/JDO is available in several editions. Your license key codifies the Kodo JPA/JDO edition you are using. Not all features are available in all editions; contact <sales@solarmetric.com> to upgrade your licenses to a more inclusive edition.

3.1. Standard Edition

The Kodo JPA/JDO Standard Edition is designed for lightweight applications that do not require enterprise features such as application server integration. The Standard Edition includes everything described in this manual except those features that explicitly state the need for a more advanced Kodo JPA/JDO edition.

3.2. Performance Pack

The Performance Pack is an add-on to the **Standard Edition**. It provides features designed to maximize the performance and scalability of your Kodo JPA/JDO applications. The Performance Pack adds the following functionality to the Kodo JPA/JDO Standard Edition:

- JDBC statement batching, described at **Section 4.9, “Statement Batching”** [440].
- Eager fetching, described at **Section 5.7, “Eager Fetching”** [473].
- Lock groups, described at **Section 5.8, “Lock Groups”** [476].
- Data and query caches, described at **Section 10.1, “Data Cache”** [560].
- Remote commit events, described at **Section 11.3, “Remote Event Notification Framework”** [587].
- The Kodo profiler, described at **Chapter 13, Profiling** [603].

3.3. Enterprise Edition

The Kodo JPA/JDO Enterprise Edition is designed for enterprise applications. It includes everything in the **Standard Edition** and the **Performance Pack**, as well as the following features:

- SQL queries, described in **Chapter 17, SQL Queries** [320].
- Integration with managed transactions, described in **Section 8.2, “Integrating with the Transaction Manager”** [535].
- XA transactions, described in **Section 8.3, “XA Transactions”** [536].
- Custom class mapping, described in **Section 7.10.1, “Custom Class Mapping”** [523].
- JPA/JDOQL subqueries, described in **Section 9.6.4, “JDOQL Subqueries”** [553].
- MethodQL, described in **Section 9.6.5, “MethodQL”** [554].
- Non-relational store access, described in **Section 9.9, “Non-Relational Stores”** [559].
- Remote `PersistenceManagers`, described in **Section 11.2, “Remote Managers”** [581].

Appendix 4. Supported Databases

Following is a table of the database and JDBC driver versions that are supported by Kodo JPA/JDO.

Table 4.1. Supported Databases and JDBC Drivers

Database Name	Database Version	JDBC Driver Name	JDBC Driver Version
Apache Derby	10.1.2.1	Apache Derby Embedded JDBC Driver	10.1.2.1
Borland Interbase	7.1.0.202	Interclient	4.5.1
Borland JDataStore	6.0	Borland JDataStore	6.0
DB2	8.1	IBM DB2 JDBC Universal Driver	1.0.581
Empress	8.62	Empress Category 2 JDBC Driver	8.62
Firebird	1.5	JayBird JCA/JDBC driver	1.0.1
Hypersonic Database Engine	1.8.0	Hypersonic	1.8.0
Informix Dynamic Server	9.30.UC10	Informix JDBC driver	2.21.JC2
InterSystems Cache	5.0	Cache JDBC Driver	5.0
Microsoft Access	9.0 (a.k.a. "2000")	DataDirect SequeLink	5.4.0038
Microsoft SQL Server	9.00.1399 (SQL Server 2005)	SQLServer	1.0.809.102
Microsoft Visual FoxPro	7.0	DataDirect SequeLink	5.4.0038
MySQL	3.23.43-log	MySQL Driver	3.0.14
Oracle	8.1,9.2,10.1	Oracle JDBC driver	10.2.0.1.0
Pointbase	4.4	Pointbase JDBC driver	4.4 (4.4)
PostgreSQL	7.2.1	PostgreSQL Native Driver	7.2 (7.2)
Sybase Adaptive Server Enterprise	12.5	jConnect	5.5 (5.5)

4.1. Apache Derby

Example 4.1. Example properties for Derby

```
kodo.ConnectionDriverName: org.apache.derby.jdbc.EmbeddedDriver
kodo.ConnectionURL: jdbc:derby:DB_NAME;create=true
```

4.2. Borland Interbase

Example 4.2. Example properties for Interbase

```
kodo.ConnectionDriverName: interbase.interclient.Driver
kodo.ConnectionURL: jdbc:interbase://SERVER_NAME:SERVER_PORT/DB_PATH
```

4.2.1. Known issues with Interbase

- Interbase does not support record locking, so datastore transactions cannot use the pessimistic lock manager.
- Interbase does not support the LOWER, SUBSTRING, or INSTR SQL functions, which means that `toLowerCase()`, `indexOf()`, and `substring()` methods in JPA/JDOQL cannot be used.

4.3. JDataStore

Example 4.3. Example properties for JDataStore

```
kodo.ConnectionDriverName: com.borland.datastore.jdbc.DataStoreDriver
kodo.ConnectionURL: jdbc:borland:dslocal:db-jdatastore.jds;create=true
```

4.4. IBM DB2

Example 4.4. Example properties for IBM DB2

```
kodo.ConnectionDriverName: com.ibm.db2.jcc.DB2Driver
kodo.ConnectionURL: jdbc:db2://SERVER_NAME:SERVER_PORT/DB_NAME
```

4.4.1. Known issues with DB2

- Floats and doubles may lose precision when stored.
- Empty char values are stored as NULL.
- Fields of type BLOB and CLOB are limited to 1M. This number can be increased by extending `DB2Dictionary`.

4.5. Empress

Example 4.5. Example properties for Empress

```
kodo.ConnectionDriverName: empress.jdbc.empressDriver
kodo.ConnectionURL: jdbc:empress://SERVER=yourserver;PORT=6322;DATABASE=yourdb
```

4.5.1. Known issues with Empress

- Empress enforces pessimistic semantics (lock on read) when not using `AllowConcurrentRead` property (which bypasses row locking) for `EmpressDictionary`.
- Only the category 2 non-local driver is supported.

4.6. Hypersonic

Example 4.6. Example properties for Hypersonic

```
kodo.ConnectionDriverName: org.hsqldb.jdbcDriver
kodo.ConnectionURL: jdbc:hsqldb:DB_NAME
```

4.6.1. Known issues with Hypersonic

- Hypersonic does not properly support foreign key constraints.
- Hypersonic does not support pessimistic locking, so non-optimistic transactions will fail unless the `SimulateLocking` property is set for the `kodo.jdbc.DBDictionary`

4.7. Firebird

Example 4.7. Example properties for Firebird

```
kodo.ConnectionDriverName: org.firebirdsql.jdbc.FBDriver
kodo.ConnectionURL: jdbc:firebirdsql://SERVER_NAME:SERVER_PORT/DB_PATH
```

4.7.1. Known issues with Firebird

- The Firebird JDBC driver does not have proper support for batch updates, so batch updates are disabled.
- Firebird does not support auto-increment columns.
- Firebird does not support the LOWER, SUBSTRING, or INSTR SQL functions, which means that `toLowerCase()`, `indexOf()`, and `substring()` methods in JPA/JDOQL cannot be used.

4.8. Informix

Example 4.8. Example properties for Informix Dynamic Server

```
kodo.ConnectionDriverName: com.informix.jdbc.IfxDriver
kodo.ConnectionURL: \
  jdbc:informix-sqli://SERVER_NAME:SERVER_PORT/DB_NAME:INFORMIXSERVER=SERVER_ID
```

4.8.1. Known issues with Informix

- none

4.9. InterSystems Cache

Example 4.9. Example properties for InterSystems Cache

```
kodo.ConnectionDriverName: com.intersys.jdbc.CacheDriver
kodo.ConnectionURL: jdbc:Cache://SERVER_NAME:SERVER_PORT/DB_NAME
```

4.9.1. Known issues with InterSystems Cache

- Support for Cache is done via SQL access over JDBC, not through their object database APIs.

4.10. Microsoft Access

Example 4.10. Example properties for Microsoft Access

```
kodo.ConnectionDriverName: com.ddtek.jdbc.sequelink.SequeLinkDriver
kodo.ConnectionURL: jdbc:sequelink://SERVER_NAME:SERVER_PORT
```

4.10.1. Known issues with Microsoft Access

- Using the Sun JDBC-ODBC bridge to connect is not supported.

4.11. Microsoft SQL Server

Example 4.11. Example properties for Microsoft SQLServer

```
kodo.ConnectionDriverName: com.microsoft.sqlserver.jdbc.SQLServerDriver
kodo.ConnectionURL: \
  jdbc:sqlserver://SERVER_NAME:1433;DatabaseName=DB_NAME;selectMethod=cursor;sendStringParametersAsUnicode=false
```

4.11.1. Known issues with SQL Server

- SQL Server date fields are accurate only to the nearest 3 milliseconds, possibly resulting in precision loss in stored dates.
- The ConnectionURL must always contain the "selectMethod=cursor" string.
- Adding `sendStringParametersAsUnicode=false` to the ConnectionURL may significantly increase performance.
- The Microsoft SQL Server driver only emulates batch updates. The DataDirect JDBC driver has true support for batch updates, and may result in a significant performance gain.
- Floats and doubles may lose precision when stored.
- TEXT columns cannot be used in queries.

4.12. Microsoft FoxPro

Example 4.12. Example properties for Microsoft FoxPro

```
kodo.ConnectionDriverName: com.ddtek.jdbc.sequelink.SequeLinkDriver
```

```
kodo.ConnectionURL: jdbc:sequelink://SERVER_NAME:SERVER_PORT
```

4.12.1. Known issues with Microsoft FoxPro

- Using the Sun JDBC-ODBC bridge to connect is not supported.

4.13. MySQL

Example 4.13. Example properties for MySQL

```
kodo.ConnectionDriverName: com.mysql.jdbc.Driver  
kodo.ConnectionURL: jdbc:mysql://SERVER_NAME/DB_NAME
```

4.13.1. Known issues with MySQL

- The default table types that MySQL uses do not support transactions, which will prevent Kodo from being able to roll back transactions. Use the `InnoDB` table type for any tables that Kodo will access.
- MySQL does not support sub-selects in versions prior to 4.1, and are disabled by default. Some operations (such as the `isEmpty()` method in a JDOQL query) will fail due to this. If you are using MySQL 4.1 or later, you can lift this restriction by setting the `SupportsSubselect=true` parameter of the `kodo.jdbc.DBDictionary` property.
- Rollback due to database error or optimistic lock violation is not supported unless the table type is one of the MySQL transactional types. Explicit calls to `rollback()` before a transaction has been committed, however, are always supported.
- Floats and doubles may lose precision when stored in some datastores.
- When storing a field of type `java.math.BigDecimal`, some datastores will add extraneous trailing 0 characters, causing an equality mismatch between the field that is stored and the field that is retrieved.
- Some version of the MySQL JDBC driver have a bug that prevents Kodo from being able to interrogate the database for foreign keys. Version 3.0.14 (or higher) of the MySQL driver is required in order to get around this bug.

4.14. Oracle

Example 4.14. Example properties for Oracle

```
kodo.ConnectionDriverName: oracle.jdbc.driver.OracleDriver  
kodo.ConnectionURL: jdbc:oracle:thin:@SERVER_NAME:1521:DB_NAME
```

4.14.1. Using Query Hints with Oracle

Oracle has support for "query hints", which are formatted comments embedded in SQL that provide some hint for how the query should be executed. These hints are usually designed to provide suggestions to the Oracle query optimizer for how to efficiently perform a certain query, and aren't typically needed for any but the most intensive queries.

Example 4.15. Using Oracle Hints

JDO:

```
Query query = pm.createQuery (...);
query.addExtension (kodo.jdbc.sql.OracleDictionary.SELECT_HINT,
    "/*+ first_rows(100) */");
List results = (List) query.execute ();
```

JPA:

```
Query query = em.createQuery (...);
query.setHint (kodo.jdbc.sql.OracleDictionary.SELECT_HINT,
    "/*+ first_rows(100) */");
List results = query.getResultList ();
```

4.14.2. Known issues with Oracle

- The Oracle JDBC driver has significant differences between different versions. It is important to use the officially supported version of the driver (10.2.0.1.0), which is backward compatible with previous versions of the Oracle server. It can be downloaded from http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/htdocs/jdbc101040.html.
- For VARCHAR fields, null and a blank string are equivalent. This means that an object that stores a null string field will have it get read back as a blank string.
- Oracle corp's JDBC driver for Oracle has only limited support for batch updates. The result for Kodo is that in some cases, the exact object that failed an optimistic lock check cannot be determined, and Kodo will throw an `OptimisticVerificationException` with more failed objects than actually failed.
- Oracle cannot store numbers with more than 38 digits in numeric columns.
- Floats and doubles may lose precision when stored.
- CLOB columns cannot be used in queries.

4.15. Pointbase

Example 4.16. Example properties for Pointbase

```
kodo.ConnectionDriverName: com.pointbase.jdbc.jdbcUniversalDriver
kodo.ConnectionURL: \
  jdbc:pointbase:DB_NAME,database.home=pointbasedb,create=true,cache.size=10000,database.pagesize=30720
```

4.15.1. Known issues with Pointbase

- Fields of type BLOB and CLOB are limited to 1M. This number can be increased by extending PointbaseDictionary.

4.16. PostgreSQL

Example 4.17. Example properties for PostgreSQL

```
kodo.ConnectionDriverName: org.postgresql.Driver
kodo.ConnectionURL: jdbc:postgresql://SERVER_NAME:5432/DB_NAME
```

4.16.1. Known issues with PostgreSQL

- Floats and doubles may lose precision when stored.
- PostgreSQL cannot store very low and very high dates.
- Empty string/char values are stored as NULL.

4.17. Sybase Adaptive Server

Example 4.18. Example properties for Sybase

```
kodo.ConnectionDriverName: com.sybase.jdbc2.jdbc.SybDriver
kodo.ConnectionURL: \
  jdbc:sybase:Tds:SERVER_NAME:4100/DB_NAME?ServiceName=DB_NAME&BE_AS_JDBC_COMPLIANT_AS_POSSIBLE=true
```

4.17.1. Known issues with Sybase

- The "DYNAMIC_PREPARE" parameter of the Sybase JDBC driver cannot be used with Kodo.
- Datastore locking cannot be used when manipulating many-to-many relations using the default Kodo schema created by the schematool, unless an auto-increment primary key field is manually added to the table.
- Persisting a zero-length string results in a string with a single space characted being returned from Sybase, Inc.'s JDBC driver.
- The BE_AS_JDBC_COMPLIANT_AS_POSSIBLE is required in order to use datastore (pessimistic) locking. Failure to set this property may lead to obscure errors like "FOR UPDATE can not be used in a SELECT which is not part of the declaration of a cursor or which is not inside a stored procedure."

Appendix 5. Common Database Errors

Following is a list of known SQL errors, and potential solutions to the problems that they represent.

Table 5.1. Known Database Error Codes

Database	Error Code	SQL State	Message	Solution
DB2	-803	23505	SQL0803N One or more values in the INSERT statement, UPDATE statement, or foreign key update caused by a DELETE statement are not valid because the primary key, unique constraint or unique index identified by "1" constrains table "%s" from having duplicate rows for those columns.	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
DB2	-511	42829	SQL0511N The FOR UPDATE clause is not allowed because the table specified by the cursor cannot be modified.	A datastore transaction read was attempted on a table that is marked as read-only. Either read the data outside of a transaction, or use optimistic transactions.
DB2	-401	42818	SQL0401N The data types of the operands for the operation ">=" are not compatible.	A mathematical comparison query was attempted on a field whose mapping was to a non-numeric field, such as VARCHAR. DB2 disallows such queries.
DB2	-302	22003	SQL0302N The value of a host variable in the EXECUTE or OPEN statement is too large for its corresponding use.	Possible attempt to store a string of a length greater than is allowed by the database's column definition. If creation is done via the mapping tool, ensure that the length attribute of the column element specifies a large enough size for the column.

Common Database Errors

Database	Error Code	SQL State	Message	Solution
DB2	-204	42S02	SQL0204N "%s" is an undefined name.	The database schema does not match the mapping defined in the metadata for the persistent class. See the mapping documentation.
DB2	-99999	22003	Numeric value out of range.	A numeric range error occurred. Ensure that the capacity of the numeric column is sufficient to hold the specified value the persistent object is attempting to store.
DB2	-99999	HY003	CLI0122E Program type out of range.	A numeric or string range error occurred. Ensure that the capacity of the numeric or string column is sufficient to store the specified value the persistent object is attempting to store.
HSQL	-8	23000	Integrity constraint violation in statement %s	Attempted modification of a row that would cause a violation of referential integrity constraints. Make sure your mapping metadata declares your database's foreign keys.
HSQL	-9	23000	Violation of unique index: 23000 Violation of unique index in statement %s	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
HSQL	-40	S1000	General error: S1000 General error java.lang.NumberFormatException: %d in statement %s	A numeric range error occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store. Note that some versions of HSQL have a bug that prevents Long.MIN_VALUE from being stored.
MySQL	1062	S1009	Invalid argument	

Common Database Errors

Database	Error Code	SQL State	Message	Solution
			value: Duplicate entry '1' for key 1	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
MySQL	1196	S1000	General error: Warning: Some non-transactional changed tables couldn't be rolled back	One or more tables that are being manipulated are not configured to be transactional. Tables in MySQL, by default, do not support transactions. Table type for schema creation can be configured with the <code>TableType</code> property of the DBDictionary configuration property.
MySQL	1213	S1000	General error: Deadlock found when trying to get lock; Try restarting transaction	A deadlock occurred during a datastore transaction. This can occur when transaction TRANS1 locks table TABLE1, transaction TRANS2 locks table TABLE2, TRANS1 lines up to get a lock on TABLE2, and then TRANS2 lines up to get a lock on TABLE1. Deadlock prevention is the responsibility of the application, or the application server in which it runs. For more details, see the MySQL deadlock documentation.
MySQL	0	08S01	Communication link failure: java.io.IOException	The TCP connection underlying the JDBC Connection has been closed, possibly due to a timeout. If using Kodo's default DataSource, connection testing can be configured via the ConnectionFactoryProperties property.
MySQL	1030	S1000	General error: Got error 139 from table hand-	This is a bug in MySQL server, and can occur when using tables of

Common Database Errors

Database	Error Code	SQL State	Message	Solution
			ler	type InnoDB when long SQL statements are sent to the server. Upgrade to a more recent version of MySQL to resolve the problem.
MySQL	1054	S0022	Column not found: Unknown column 'Infinity' in 'field list'	MySQL disallows storage of <code>Double.POSITIVE_INFINITY</code> or <code>Double.NEGATIVE_INFINITY</code> values.
Oracle	17069	null	Use explicit XA call	Manual transaction operations were attempted on a data source that was configured to use an XA transaction. In order to utilize XA transactions, set the kodo.ConnectionFactoryMode property to managed.
Oracle	17433	null	invalid arguments in call	The Oracle JDBC driver throws this exception when a null username or password are specified. A username and password was not specified in the Kodo configuration, nor was it specified in the database configuration mechanism, nor was it specified in the <code>PersistenceManager.getPersistenceManager</code> invocation.
Oracle	904	42000	ORA-00904: invalid column name	The database schema does not match the mapping defined in the metadata for the persistent class. See the mapping documentation.
Oracle	1722	42000	ORA-01722: invalid number	A number that Oracle cannot store has been persisted. This can happen when a string field in the persistent class is mapped to an Oracle column of type NUMBER

Common Database Errors

Database	Error Code	SQL State	Message	Solution
				and the String value is not numeric.
Oracle	1000	72000	ORA-01000: maximum open cursors exceeded	<p>Oracle limits the number of statements that can be open at any given time, and the application has made requests that keep open more statements than Oracle can handle. This can be resolved in one of the following ways:</p> <ol style="list-style-type: none"> 1. Increase the number of cursors allowed in the database. This is typically done by increasing the <code>open_cursors</code> parameter in the <code>initSID-NAME.ora</code> file. 2. Ensure that Kodo query results and Extent iterators are being closed, since open results will maintain an open <code>ResultSet</code> on the server side until they are garbage collected. 3. Decrease the value of the <code>Max-CachedStatements</code> parameter in the ConnectionFactory-Properties configuration property.
Oracle	932	42000	ORA-00932: inconsistent datatypes: expected - got CLOB	<p>A normal string field was mapped to an Oracle CLOB type. Oracle requires special handling for CLOBs. Ensure that the column for this field specifies a <code>jdbc-type</code> of <code>clob</code> or a length of <code>-1</code>.</p>

Common Database Errors

Database	Error Code	SQL State	Message	Solution
Oracle	1	23000	ORA-00001: unique con- straint (%s) vi- olated	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
Oracle	0	null	Underflow Excep- tion	A numeric underflow occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store. Note that Oracle NUMERIC fields have a limitation of 38 digits.
Oracle	0	null	Overflow Excep- tion	A numeric underflow occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store. Note that Oracle NUMERIC fields have a limitation of 38 digits.
Pointbase	78003	ZW003	The value "%s" cannot be con- verted to a num- ber.	This can happen when a string field in the persistent class is mapped to a numeric column, and the string value cannot be parsed into a number.
Pointbase	25203	22003	Data exception - - numeric value out of range. %d.	A numeric range error occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store.
PostgreSQL	0	null	ERROR: Unable to identify an op- erator '>=' for types 'numeric' and 'double pre- cision' You will have to retype this query using an explicit cast	An integer field is mapped to a decimal column type. PostgreSQL disallows performing numeric comparisons between integers and decimals.

Common Database Errors

Database	Error Code	SQL State	Message	Solution
PostgreSQL	0	null	ERROR: Cannot insert a duplicate key into unique index bug488pcx_pkey	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.
PostgreSQL	0	null	ERROR: Attribute 'infinity' not found	PostgreSQL disallows storage of Double.POSITIVE_INFINITY or Double.NEGATIVE_INFINITY values.
PostgreSQL	0	null	You will have to retype this query using an explicit cast	A string field is mapped to a numeric column type. PostgreSQL disallows performing string comparisons in queries against a numeric column.
SQLServer	0	08007	Can't start a cloned connection while in manual transaction mode.	Append ";SelectMethod=cursor" to the ConnectionURL. See the description of the problem on the Microsoft support site .
SQLServer			sp_cursorclose: The cursor identifier value provided (abcdef0) is not valid.	This can sometimes show up as a warning when Kodo is closing a prepared statement. It is due to a bug in the SQLServer driver, and can be ignored, since it should not affect anything.
SQLServer	306	HY000	The text, ntext, and image data types cannot be compared or sorted, except when using IS NULL or LIKE operator.	A query ordering was attempted on a field that is mapped to a CLOB or BLOB, which is disallowed by SQLServer.
SQLServer	8114	HY000	Error converting data type varchar to %s.	This can happen when a string field in the persistent class is mapped to a numeric column, and the string value cannot be parsed into a number.

Common Database Errors

Database	Error Code	SQL State	Message	Solution
SQLServer	245	22018	Syntax error converting the varchar value '%s' to a column of data type int.	This can happen when a string field in the persistent class is mapped to a numeric column, and the string value cannot be parsed into a number.
SQLServer	2627	23000	Violation of PRIMARY KEY constraint 'PK_%s'. Cannot insert duplicate key in object '%s'.	Duplicate values have been inserted into a primary key column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate primary keys when using application identity.
SQLServer	169	HY000	A column has been specified more than once in the order by list. Columns in the order by list must be unique.	Ensure that there are no duplicates in the ordering of the query.
SQLServer	0	HY000	Object has been closed.	The TCP connection underlying the JDBC connection may have been closed, possibly due to a timeout. If using Kodo's default DataSource, connection testing can be configured via the ConnectionFactoryProperties configuration property.
Sybase	311	ZZZZZ	The optimizer could not find a unique index which it could use to scan table '%s' for cursor 'jconnect_implicit_%d'	A pessimistic lock was attempted on a table that does not have a primary key (or other unique index). By default, the Kodo mapping tool does not create primary keys for join tables. In order to use datastore locking for relations, an IDENTITY column should be added to any tables that do not already have them.
Sybase	2762		The 'CREATE TABLE' command is	This may happen when

Common Database Errors

Database	Error Code	SQL State	Message	Solution
			not allowed within a multi-statement transaction in the 'tempdb' database.	running the schema tool against a Sybase database that is not configured to allow schema-altering commands to be executed from within a transaction. This can be enabled by entering the command sp_dboption database_name,"ddl in tran", true from isql . See the Sybase documentation for allowing data definition commands in transactions .
Sybase	0	JZ0BE	JZ0BE: BatchUpdateException: Error occurred while executing batch statement: Arithmetic overflow during implicit conversion of NUMERIC value '%d' to a NUMERIC field.	A numeric overflow occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store.
Sybase	0	JZ00B	JZ00B: Numeric overflow.	A numeric overflow occurred. Ensure that the capacity of the numeric column is sufficient to store the specified value the persistent object is attempting to store.
Sybase	257	42000	Implicit conversion from data-type 'VARCHAR' to 'TINYINT' is not allowed. Use the CONVERT function to run this query.	A string field is stored in a column of numeric type. Sybase disallows querying against these fields.
Sybase	169	ZZZZZ	Expression '1' and '8' in the ORDER BY list are same. Expressions in the ORDER BY list must be unique.	Ensure that there are no duplicates in the ordering of the query.
Sybase	511	ZZZZZ	Attempt to update or insert row failed because resultant row of size 2009 bytes is larger	Possible attempt to store a string of a length greater than is allowed by the database's column definition. If creation is

Common Database Errors

Database	Error Code	SQL State	Message	Solution
			than the maximum size (1961 bytes) allowed for this table.	done via the mapping tool, ensure that the length attribute of the column element specifies a large enough size for the column.
Sybase	2601	23000	Attempt to insert duplicate key row in object '%s' with unique index '%s'	Duplicate values have been inserted into a column that has a UNIQUE constraint. It is the responsibility of the application to deal with prevention of insertion of duplicate values.

Appendix 6. Upgrading Kodo

This document describes how to upgrade to a new version of Kodo from the prior version. When skipping versions, please follow the steps outlined as though you are converting to each intermediate version.

6.1. Compatibility Configuration

The sections below detail specific steps to switch from one Kodo release to another. Kodo also offers backwards compatibility settings for certain runtime behaviors, encompassed in the `kodo.Compatibility` property. This property uses Kodo's **plugin syntax** to control the following options:

- `ValidateFalseReturnsHollow`: Whether to return hollow objects (objects for which no state has been loaded) from calls to `PersistenceManager.getObjectById(oid, false)`. This is the default behavior of Kodo 4.0 and above. Previous Kodo versions, however, always loaded the object from the datastore. Set this property to `false` to get the old behavior.
- `ValidateTrueChecksStore`: Prior to Kodo 4.0, calling `PersistenceManager.getObjectById(oid, true)` always checked the datastore to be sure the given oid existed, even when the corresponding object was already cached. Kodo 4.0 and above does not check the datastore when the instance is already cached and loaded. Set this property to `true` to mimic previous behavior.
- `CopyObjectIds`: Kodo versions prior to 3.0 always copied oid objects before returning them to you. This prevents possible errors resulting from you mutating the oid object by reference, but wasn't very efficient for the majority of users who don't modify oid instances. Thus Kodo 3.0 and higher does not copy oids by default. Set this property to `true` to force Kodo to copy.
- `CloseOnManagedCommit`: If `true`, then the JDO `PersistenceManager` will close after a managed transaction commits, assuming you have invoked the `close` method. If this is set to `false`, then the `PersistenceManager` will not close. This means that objects passed to a processing tier in the same JVM will still be usable, as their owning `PersistenceManager` will still be open. This behavior is not in strict compliance with the JDO specification, but is convenient for applications that were coded against Kodo 2.x, which did not close the `PersistenceManager` in these situations. The default for this property is `true`, meaning that the `PersistenceManager` will close properly.
- `QuotedNumbersInQueries`: Whether to interpret quoted numbers in query strings as numbers, as opposed to strings. Defaults to treating quoted numbers as strings. Set to `true` to treat quoted numbers as numbers instead to mimic the behavior of Kodo 3.1 and prior.
- `StrictIdentityValues`: Whether to require identity values used for finding application identity instances to be of the exact right type. By default, Kodo allows stringified identity values, and performs conversions between numeric types.

For example, the setting below cause Kodo to copy oids and not to return hollow objects:

JPA XML format:

```
<property name="kodo.Compatibility"
  value="ValidateFalseReturnsHollow=false, CopyObjectIds=true">
```

JDO properties format:

```
kodo.Compatibility: ValidateFalseReturnsHollow=false, CopyObjectIds=true
```

6.2. Migrating from Kodo 3.0 to Kodo 3.1

The following are points to consider when upgrading from Kodo 3.0.x to Kodo 3.1.0:

- Kodo now uses its own logging framework by default instead of the Commons Logging framework. To use the Commons Logging framework as in older versions of Kodo, set the `kodo.Log` property to `commons`. See [Chapter 3, Logging \[417\]](#) for details.
- The `DBDictionary` now supports the `DefaultSchemaName` parameter. See [Section 4.4, “Database Support” \[428\]](#). The default is `null`, which will cause Kodo to check all schemas for unqualified tables. Setting this will cause Kodo to only check the specified schema for unqualified tables. `kodo.jdbc.Schemas` is now only used in conjunction with reverse-mapping processes to identify which schemas to examine for reverse mapping.

6.3. Migrating from Kodo 3.1 to Kodo 3.2

The following are points to consider when upgrading from Kodo 3.1.x to Kodo 3.2.0:

- Configuration of the management and profiling capabilities has been simplified. All configuration now happens through the `kodo.ManagementConfiguration` property. For more information, please see [Chapter 12, Management and Monitoring \[591\]](#) and [Chapter 13, Profiling \[603\]](#).

6.4. Migrating from Kodo 3.x to Kodo 4.0

Kodo 4 is a major release. Kodo 4 supports both the JPA and JDO specifications on top of the same persistence kernel. As such, Kodo's package structure and configuration options have changed to reflect its kernel-based architecture. Additionally, Kodo 4 incorporates the latest JDO specification's relational mapping format, superseding Kodo 3's proprietary mapping options.

Kodo 4 does include backwards compatibility for Kodo 3's primary interfaces, and for Kodo 3's metadata and mapping files. The following sections describe how to use these compatibility features.

6.4.1. API Compatibility

In addition to the standard JDO APIs, the following Kodo 3 interfaces are available in Kodo 4. The Kodo 4 equivalent of each interface is shown in parentheses.

- `kodo.runtime.KodoPersistenceManagerFactory` (`kodo.jdo.KodoPersistenceManagerFactory`)
- `kodo.runtime.KodoPersistenceManager` (`kodo.jdo.KodoPersistenceManager`)
- `kodo.runtime.KodoExtent` (`kodo.jdo.KodoExtent`)
- `kodo.runtime.KodoHelper` (`kodo.jdo.KodoJDOHelper`)
- `kodo.runtime.FetchConfiguration` (`kodo.jdo.KodoFetchPlan`)
- `kodo.runtime.SequenceGenerator` (`javax.jdo.datastore.Sequence`)
- `kodo.query.KodoQuery` (`kodo.jdo.KodoQuery`)
- `kodo.jdbc.runtime.JDBCPersistenceManagerFactory` (`kodo.jdo.PersistenceManagerFactoryImpl`)

- `kodo.jdbc.runtime.JDBCFetchConfiguration` (`kodo.jdo.jdbc.JDBCFetchPlan`)

To continue to use these Kodo 3 interfaces in your application, simply keep your `javax.jdo.PersistenceManagerFactoryClass` property set to its Kodo 3 value of `kodo.jdbc.runtime.JDBCPersistenceManagerFactory`. These backwards-compatibility interfaces even extend the corresponding Kodo 4 components, so you can access any new Kodo 4 APIs or slowly migrate portions of your code to use Kodo 4 rather than Kodo 3 interfaces.

Note

If you bootstrap your Kodo 3 application using something other than the `JDOHelper` or `KodoHelper` bootstrapping methods, you will have to manually construct a `kodo.jdbc.runtime.JDBCPersistenceManagerFactory` from a Kodo 4 factory.

6.4.2. Metadata and Mappings

To use Kodo 3 metadata and mapping files, you must set the following configuration properties:

- `kodo.MetaDataFactory`: `kodo3`
- `kodo.jdbc.MappingFactory`: `file` if you use `.mapping` files, `metadata` if you use mapping metadata extensions, or `db` if you use a database table to hold your mapping data.

Note

If you'd like to switch to Kodo 4 metadata and mappings at any time, you can do so using the mapping tool. [Section 7.5.1, “Importing and Exporting Mapping Data” \[505\]](#) covers this process in detail. Be sure to pass the `-meta true` flag to the tool to migrate your metadata along with your mappings.

Kodo 3 used a table called `JDO_SEQUENCE` to generate sequence numbers by default. Kodo 4 uses a table called `KODO_SEQUENCE_TABLE` instead. To continue using your `JDO_SEQUENCE` table, set the following property:

- `kodo.Sequence`: `Table=JDO_SEQUENCE`

Finally, Kodo 4 defaults to the *dynamic* schema factory, meaning it will not reflect on your schema to detect foreign keys (JDO's new relational mapping metadata allows you to specify foreign keys in your mapping, so Kodo 4 does not have to rely on schema reflection). If you continue to use Kodo 3 mappings and your schema has foreign key constraints Kodo must meet, set the following:

- `kodo.jdbc.SchemaFactory`: `native(ForeignKeys=true)`

6.4.3. Configuration

Many configuration properties and defaults have changed. Like previous Kodo releases, Kodo 4 will issue warnings if you attempt to set configuration options that it does not recognize. Refer to [Chapter 2, Configuration \[393\]](#) for a complete listing of Kodo 4's configuration properties and defaults. Also, be sure to read [Section 6.1, “Compatibility Configuration” \[648\]](#) above to learn about Kodo's special compatibility options.

Appendix 7. Development and Runtime Libraries

- `kodo.jar`: Library for development of applications for Kodo JPA/JDO.
- `kodo-runtime.jar`: Library for runtime use of Kodo JPA/JDO.
- `kodo-api.jar`: Published Kodo APIs that can be used for build-time dependency resolution.
- `kodo-wl81manage.jar`: Library required to connect to WebLogic 8.1 with `remotejmxtool`. This jar must be included in the WebLogic system classpath.
- `jca1.0.jar`: Java Connector Architecture, used internally by Kodo. Required for all development and runtime use of Kodo. See <http://java.sun.com/j2ee/connector/>.
- `jdbc-hsql-1_8_0.jar`: Hypersonic pure-java database engine. It is used as a simple JDBC driver for learning Kodo, but it is not required for general development or runtime use. See <http://hsqldb.sourceforge.net/>. This jar is a JDK 1.3 compile which doesn't support JDK 1.4 and higher features found in the standard distribution.
- `jdbc2_0-stdext.jar`: Standard extensions to the JDBC2 API. Required for all development and runtime use of Kodo, unless JDK version 1.4 or higher is being used (since the library is included with JDK 1.4). See <http://java.sun.com/products/jdbc/>.
- `jline.jar`: Console handling utility. Used by the `sqlline.jar` file. This file is only used for SQL debugging; it never needs to be included for Kodo runtime operations. See [Section 4.15, “The SQLLine Utility” \[449\]](#) for more details. Distributed from <http://jline.sourceforge.net>.
- `jpa.jar`: Interfaces defined by the Java Persistence Architecture in J2EE 5 standard. Required for all development and runtime use of JPA APIs.
- `jdo.jar`: Interfaces defined by the Java Data Objects standard. Required for all development and runtime use of JDO APIs.
- `jndi.jar`: JNDI interfaces. Required for all development and runtime use of Kodo with JNDI, unless JDK 1.4 or higher is being used. See <http://java.sun.com/products/jndi/>.
- `jta-spec1_0_1.jar`: JTA interfaces. Required all development and runtime use of Kodo. See <http://java.sun.com/products/jta/>.
- `sqlline.jar`: Command line utility for executing SQL commands directly against a database. This file is only used for SQL debugging; it never needs to be included for Kodo runtime operations. See [Section 4.15, “The SQLLine Utility” \[449\]](#).
- `xalan.jar`: **Apache** implementation of the transformation parts of the Java API for XML Parsing. Any JAXP-compliant implementation can be used in place of `xalan.jar`. If JDK version 1.4 or higher is being used, this file is optional, since JDK 1.4 includes a built-in JAXP-compliant transformer. See <http://xml.apache.org/xalan-j/>.
- `xercesImpl.jar`: **Apache** implementation of the Java API for XML Parsing. Any JAXP-compliant implementation can be used in place of `xerces.jar`. If JDK version 1.4 or higher is being used, this file is optional, since JDK 1.4 includes a built-in JAXP-compliant parser. See <http://xml.apache.org/xerces-j/>.
- `xml-apis.jar`: XML interfaces. Required for Kodo development and runtime, unless JDK 1.4 or higher is being used.

Appendix 8. Release Notes

4.0.1 - ???

- Bugfixes
 - Fixed a `NullPointerException` from the enhancer when a property access entity has a non-trivial getter method and is missing a setter method.

4.0.0 - May 14, 2006

- New features

Kodo 4 is a major release. Its primary new features are support for the Java Persistence API and Java Data Objects atop a shared persistence kernel, and a new mapping system to ease the creation of custom mappings. However, Kodo 4 includes a host of other improvements as well. See the documentation for details.

- Notable Changes

- Major changes to internal APIs, package structure, mapping format, and configuration options. See [Section 6.4, “Migrating from Kodo 3.x to Kodo 4.0” \[649\]](#) for backwards-compatibility options.
- The Oracle dictionary will now automatically test the batching capabilities of the JDBC driver when it first connects and enable or disable batching support as a result of the test. See the `TestBatchSupport` parameter description.
- The included Hypersonic database has been upgraded from version 1.7.0 to 1.8.0.
- The supported version of Apache Derby is now upgraded to the non-alpha release 10.1.1.0. See [Section 4.1, “Apache Derby” \[629\]](#).
- Calling `makeTransactional` on an instance will now force a version check for the instance at commit time, even if the instance hasn't changed during the transaction.
- The official supported version of the Microsoft SQLServer driver is now 1.0.809.102.
- The official supported version of the Oracle JDBC driver is now 10.2.0.1.0.

3.4.0 - November 8, 2005

- New features

- Added a new [Section 2.7.40, “kodo.ManagedRuntime” \[406\]](#) option for improved integration with BEA WebLogic Server. It is used by default when using WLS.
- Added the `AttachFetchFields` parameter of the `kodo.PersistenceManagerImpl` property to enable the automatic addition of detach fields to the current fetch configuration. See [Section 11.1.3, “Defining the Detached Object Graph” \[578\]](#).
- Query cache now caches aggregates and projections, including queries that use grouping. Query cache also caches queries that compare single-valued fields to persistence-capable parameters (such as `select from Employee where address == :addr`).

- Added support for savepoints, with both in-memory and JDBC 3 plugins. Note that some drivers / database combinations do not support JDBC 3 or savepoints. For more details, see [Section 9.5, “Savepoints” \[548\]](#)
- Added support for Oracle Object fields through JDBC SQLData interfaces.
- `JDODataStoreExceptions` will now contain the failed persistent instance in the `getFailedObject` whenever possible. Enhancement #1183.
- When attaching multiple instances, object lookups will be batched together to make the process of merging in multiple instances more efficient.
- `SQLWarning` instances are now checked and can be handled in a customizable way. The `WarningAction` parameter of the `kodo.ConnectionFactoryProperties` property can be used to control this.
- Parameters to `PreparedStatements` will now be tracked so that they can be included in error messages if the statement fails. The `TrackParameters` parameter of the `kodo.ConnectionFactoryProperties` property can be used to control this.
- The `PropertiesReverseCustomizer` will now log messages to the TRACE level of the `Metadata` channel reporting whether or not a custom name was found for each class and field name that is reverse mapped.
- It is now possible to specify the default SQL type name mapping to java types with the `typeMap` argument to the reverse mapping tool. See [Section 7.2, “Reverse Mapping” \[495\]](#).
- Kodo JDO Enterprise Edition and the Kodo JDO Performance Pack now include a query cache plug-in that uses Tangosol Coherence cache products. See the [data cache integration documentation](#) for more information regarding caching. If you make use of Tangosol for both your query and datacache you no longer need to specify a remote commit provider.
- Kodo now logs its runtime configuration to the `kodo.Runtime` channel at TRACE level when the `PersistenceManagerFactory` initializes.
- Added the `UseSetFormOfUseForUnicode` parameter to the `DBDictionary` for Oracle, which enables Kodo to perform special configuration for unicode fields required for Oracle versions prior to 10i. See [OracleDictionary.UseSetFormOfUseForUnicode](#).
- Kodo management capability now supported in JBoss 4.
- Notable Changes
 - Added ability for `SQLServerDictionary` to treat `UNIQUEIDENTIFIER` as either `String` or `byte[]` data.
 - Official Oracle JDBC driver support has been upgraded from 9.0.1.0.0 to 10.1.0.4. See [Appendix 4, Supported Databases \[629\]](#).
 - Kodo now determines whether or not an externalized field should be part of the default fetch group based on the type that the field is externalized to, rather than the actual field type. This means that fields of unknown type (which would normally be excluded from the default fetch group by default) will be included in the initial fetching if they are externalized to a primitive type that Kodo understands natively.
 - Eager query result lists now remain open after the `PersistenceManager` has closed.
 - Attempts to enhance JDK1.5 enum types will result in an error at enhancement time, rather than deferred unexpected behavior later in execution. Kodo does support fields of type enum in managed instances, but not managed enums themselves.
- Bugfixes
 - A bug with queries involving quoted numbers has been fixed. In Kodo 3.1 and prior, the first digit of a single-quoted character literal of any length was treated as a character constant, and double-quoted numbers were treated as unquoted nu-

meric constants. Kodo 3.2 and 3.3 were undeterministic regarding the handling of numeric comparisons to single- and double-quoted literals. By default, Kodo 3.4 follows the JDO 2.0 rules for quoted numeric literals: double-quoted and single-quoted single-character literals are treated as character constants, and double-quoted and single-quoted multi-character literals are invalid in numeric comparisons. The old 3.1 behavior can be emulated by adding `QuotedNumber=InJDOQL=3.1` to the `kodo.Compatibility` configuration parameter.

- Improved application id tool to be more JDK 1.5 compatible. Changes simple static class reference into a full `Class.forName` call to register class in VM.
- Positional parameters for SQL queries started at 0; now correctly start at 1 (0-based indexing is still supported).
- Query reselut projections including null-valued one-to-one relations with queries that allow nulls will be properly retrieved.
- The management function no longer directly references classes found only in the `kodo-jdo.jar`. This caused `java.lang.NoClassDefFoundErrors` in some configurations. Bug 1197.

3.3.4 - July 22, 2005

- Notable Changes
 - Pessimistic locks on embedded persistent types are now obtained at the same time as locks on their owning records (and vice versa), reducing the number of database roundtrips required to lock a instance and its embedded record(s).
 - Improved performance of getting `PersistenceManagers` when the `kodo.PersistentClasses` property is set.
- Bugfixes
 - The Kodo Workbench will be much faster when mounting many classes. Bug 1180.
 - The `SequenceGenerator.ensureCapacity()` method now uses the passed-in capacity number instead of the configured generator increment.
 - Fixed bug that cleared state of new instances that were not changed within the transaction when `RestoreValues` was true.
 - Removed extraneous `DISTINCT` from some eager fetching selects.
 - When using a third-party data source that is integrated with a JTA transaction manager, calls to `KodoPersistenceManagerFactory.getPersistenceManager(boolean managed, int retainMode)` will now return a `PersistenceManager` that uses `ConnectionFactory2` instead of the primary (managed) connection factory.
 - Fixed issues with tokenization of strings in certain appidtool generated classes.

3.3.3 - May 20, 2005

- New features
 - Added `DriverDeserializesBlobs` property to the `MySQLDictionary`. Most MySQL drivers automatically deserialize BLOBs on calls to `ResultSet.getObject`, but some do not.
 - Kodo JDO Enterprise Edition and the Kodo JDO Performance Pack now includes a cache plug-in that supports GemStone's GemFire cache products. See the [data cache integration documentation](#) for details.

- Notable Changes
 - `com.solarmetric.profile.ProfilingHelper` and assorted classes were renamed to `ExecutionContext-NameProvider` to better reflect the role that this interface plays in the Kodo profiling framework.
 - Query cache now considers parameters when calculating query key hash code (parameters were always part of equals comparisons).
- Bugfixes
 - The Kodo Workbench now correctly uses any classpath elements (such as JDBC driver jars) that you add to the Workbench when opening a query browser.
 - Fields beginning with multiple underscore characters will have all leading underscores properly trimmed. Bug 1125.
 - Fixed query caching problem with queries that involve subqueries. Bug 1122.
 - Fixed query cache interaction with certain remote commit provider configurations. Bug 1130.
 - Fixed potential exception when detaching an application identity class with embedded fields in the configured fetch groups. Bug 1142.
 - Fixed problem where in memory JDOQL queries using `String.matches` might match a value when it should not. Bug 1143.
 - Corrected case where using `this instanceof X` in a query whose candidate class is horizontally mapped would not limit the results to class X. Bug 1123.

3.3.2 - April 14, 2005

- Bugfixes
 - Fixed problem with base-horizontal-vertical hierarchy when using application identity. Bug 1114.
 - Fixed problem with `ConcurrentModificationException` in query cache. Bug 1115.
 - Fixed problem with ref-constant non-standard joins when using a negative number. Bug 1116.
 - Fixed problem with deadlock in datastore cache. Bug 1120.

3.3.1 - March 28, 2005

- Notable Changes
 - Throwing a `JDOFatalException` from a `TransactionListener` now propagates the exception to the user. The exception will cause a rollback if it occurs during a commit.
- Bugfixes
 - Corrected an inefficiency when performing a query against an entire hierarchy of mapped persistence-capable classes, where the candidate class extended a horizontally-mapped type.
 - Fixed problem with timestamps reporting as unknown column type on Oracle 10 with certain combination of driver and database. Bug 1111.
 - Fixed bug in new raw SQL handling abilities of `Row` class.

- Fixed bug preventing Kodo from reflecting on all but the first schema-qualified table in the schemas list.
- Corrected possible `NullPointerException` when committing a remote `PersistenceManager` transaction containing newly-persisted application identity objects in `Set` or `Map` fields.
- Fixed bug in profiler where processing events could result in a `NullPointerException`.
- Added setting `UseClobs` to allow for MySQL to allow clob use on versions which handle this correctly. Defaults to false for compatibility. Bug 1109.

3.3.0 - March 15, 2005

- New features
 - New `KodoPersistenceManager.setPopulateDataCache` method that allows transactions to control whether objects accessed during the transaction will fill the data cache. See the **Javadoc**.
 - New `KodoPersistenceManager.setLargeTransaction` method that allows transactions that act upon a large number of objects to use less memory if periodic flushes are performed. See the **Javadoc**.
 - Added preview support for **single field identity**, a JDO 2 feature. This allows for the use of application identity without writing object id classes when using a single primary key field.
 - Added lifecycle event listening framework as a JDO 2 preview feature. This currently requires importing `kodo.event` instead of `javax.jdo`.
 - Added preview support for `PersistenceManager.getObjectById(Class cls, Object value)`. This convenience method retrieves instances based on oid, primary key value, and stringified oids.
 - Added ability to dynamically generate data structs used for datacache and remote use. Classes are dynamically created to avoid primitive wrappers, optimizing memory use and load/store performance. See the **kodo.DynamicDataStructs** configuration property for more details.
 - Added ability to have caches evict based on schedule. Kodo's default caches can now parse "cron" style scheduling strings to evict at granularity from minute to month. See **Section 10.1.1, "Data Cache Configuration" [561]**.
 - New `KodoPersistenceManager.preFlush` method runs pre-flush actions such as persistence-by-reachability, deletion of dereferenced dependent objects, instance callbacks, and inverse relationship management without flushing. See the **Javadoc**.
 - New metadata extensions allow definition of the JDBC type or SQL type name for single column mappings (i.e. value mappings).
 - Numeric fields can now have their value assigned from the sequence generator for the owning class. This is especially useful for assigning application identity primary key values.
 - New mapping tool argument allows tool to create SQL scripts rather than XML schema files or directly acting on the database. See **Section 7.1, "Forward Mapping" [489]**.
 - Kodo now works with JMX 1.2 implementations, including those that implement JSR 160. Configuration of the management capability has changed slightly. See **Chapter 12, Management and Monitoring [591]**.
 - Apache Derby is now a supported database. See **Section 4.1, "Apache Derby" [629]**.
 - Added support for `KodoPersistenceManager.refreshAll(JDOException)`.

- Enhanced `JMSRemoteCommitProvider` with options for passing properties to the JNDI `InitialContext` and for attempting to reconnect to the JMS topic if the JMS system notifies Kodo of a serious connection error. See [Section 11.3.1, “Remote Commit Provider Configuration” \[587\]](#).
- Added ability to insert raw SQL into a given `kodo.jdbc.sql.Row`. This is useful for performing server side functions to generate values for custom field mappings.
- Fields of type `Collection` and `Map` that use Java 5 generics no longer need JDO metadata specifying collection element type or map key / value types.
- Java 5 enum field types can now be persisted natively (without the use of an externalizer). Such fields must still be listed in the JDO metadata file as `persistence-modifier="persistent"`, as enums are not among the spec-endorsed persistent field types in JDO1. Collections of enums and maps with enum keys or values are not yet supported.
- Kodo now provides a syntax for specifying additional lock groups in subclasses that are not used in the corresponding least-derived types. See [Section 5.8.1, “Lock Groups and Subclasses” \[477\]](#) for details.
- Notable Changes
 - Sequences assigned with the `db` and `db-class` sequence factories will now ensure that the assigned sequence value is at least 1 in order to be able to distinguish between auto-assigned values and cases where the default value is 0 for a primary key field.
 - Moved and repackaged Jakarta Commons and RegExp libraries internally. This removes the dependency on the Jakarta jars as well as avoids issues with conflicting library versions. Note that to use Commons Logging plugin instead of Kodo's default one, you still need to include the Commons Logging jar in your classpath.
 - The API of Kodo's internal datastore identity type, `kodo.util.Id` has been aligned with JDO 2 single field identity types.
 - Simplified and enhanced the datastore cache. Standard APIs like `DataCache.pin` and `DataCache.remove` are the same, but some less-used APIs have changed. See the Javadoc for the `kodo.datacache` package for details.
 - Changed the handling of object id assignment. Now, `PersistenceManager.getObjectId(pc)` and `JDOHelper.getObjectId(pc)` always return the final object id of the given instance. If the instance is new and uses auto-increment columns for its identity, these methods will cause a flush so that the identity value(s) can be determined. If the instance uses application identity, you cannot change any primary key fields after retrieving the object id.
 - Removed the `kodo.AutoIncrementConstraints` configuration property. Kodo now determines whether it needs to perform auto increment tracking dynamically.
 - The `kodo.jdbc.schemas` property is now used to limit the tables visible during runtime schema validation in addition to its traditional use to limit schema reflection in Kodo command-line tools.
 - Altered the behavior of the `detach` methods when the `RollbackOnly` flag is set to allow multiple attach/modify/detach/rollback cycles. See [Section 11.1.1, “Detach Behavior” \[577\]](#) for details.
 - The Coherence datacache plug-in now performs named cache lookups using the `com.tangosol.net.CacheFactory.getCache(String)` method by default. This differs from earlier Kodo versions, in which the `CacheFactory.getDistributedCache(String)` method was used by default. This change simplifies the configuration of the integration with Coherence, and allows use of Coherence's near cache capabilities, which were not previously accessible via Kodo's out-of-the-box configuration.
 - `StoreManager.newDataStoreId()` signature has been changed to optimize access to non-string types.
 - Aligned single-string JDOQL with most recent version of JDO 2 draft. Kodo still supports its previous single-string grammar, but see [Section 11.9, “Single-String JDOQL” \[248\]](#) for the grammar you should use going forward.
 - In support of the distributed data cache framework the `TCPRemoteCommitProvider` has been improved to more effi-

cently handle high loads. Socket connections are pooled and reused, event messages are more compact, and properties for tuning performance (pool size and worker threads) have been added. See [Section 11.3.1.2, "TCP" \[588\]](#).

- Bugfixes
 - Added "CrossJoinClause" (??? [431]), "InnerJoinClause" (??? [431]), "OuterJoinClause" (??? [431]), and "RequiresConditionForCrossJoin" (??? [431]), in order to allow the customization of join clauses. See the description for bug #1103.
 - Made persisting many new instances in a remote persistence manager more efficient. Bug 1023.
 - In-memory queries involving `java.util.Date` fields work again. Bug 1057.
 - Queries involving non-managed parameters now do not have a side effect of making the parameter transactional. Bug 1061.
 - Fixed problem where `kodo.rar` does not deploy under JBoss 4.0. Bug 1063.
 - Fixed bug with queries using range and DISTINCT on SQLServer via changes to `DBDictionary`. Bug 1080.
 - Aggregate queries are no longer issued FOR UPDATE.

3.2.4 - January 6, 2005

- Bugfixes
 - Fixed bug with unique queries and query cache interaction.
 - Fixed query cache bug in which querying uncommitted changes in one transaction could cause `JDOObjectNotFoundException` in another persistence manager.
 - Kodo can now connect to JBoss 3.2.5 via JMX.

3.2.3 - November 18, 2004

- New features
 - Added `MaximizeBatchSize` configuration option to default `kodo.jdbc.UpdateManager` property. Defaults to `true`, indicating that Kodo should sort statements in order to optimize batch size when statement batching is on.
- Notable Changes
 - `kodo.util.ProxyCollection` and `kodo.util.ProxyMap` no longer implement `java.util.Collection` and `java.util.Map` (respectively). This allows people to implement their own `ProxyCollection` in JDK 1.5 without compiler errors relating to generics.
- Bugfixes
 - Enabled ability to force no class indicator during reverse mapping process by specifying "none" as `kodo.jdbc.ClassIndicator`.
 - When Kodo encounters errors while processing a registered persistent type for the first time, it can now log a warning and retry the registration later instead of throwing an error. See [Section 2.7.57, "kodo.RetryClassRegistration" \[410\]](#).

3.2.2 - October 15, 2004

- New features
 - Added ability to receive a callback when Kodo discovers an orphaned database key, with built-in options for logging a message, throwing an exception, or doing nothing. See **Section 7.11, “Orphaned Keys” [525]**.
 - Added ability to control JBuilder logging verbosity.
- Bugfixes
 - Fixed problem where reverse mapping the same classes twice in the Kodo Workbench would have issues when reloading the classes.
 - Fixed problem where setting the persistence-modifier of a field to "none" in the Kodo Workbench would not be preserved.
 - Prevent the connection pool from thinking that it is out of connections after repeated failed attempts to connect.
 - Fixed a bug in which conditions limiting a SELECT to certain subclasses could be left out when using final subclasses or the `kodo.PersistentClasses` property.
 - Fixed a bug keeping collection and map fields from being detached during detach-on-close.
 - Changed class-criteria constrained one to many fields to not null all back references when there was no inverse owner. The implicit inverse needs to be manually set to null in these cases.

3.2.1 - October 5, 2004

- Notable Changes
 - Changed the semantics of the `javax.jdo.option.RestoreValues` and `javax.jdo.option.RetainValues` properties to match JDO 1.0.1 specification. Previously, rollbacks with `RestoreValues` set but `RetainValues` unset would preserve the dirty state of the instances. They now transition to hollow. Conversely, previous rollbacks with `RestoreValues` unset but `RetainValues` set would transition the instances to hollow. They now rollback their state and transition to persistent-nontransactional.
 - Management and profiling of Kodo within WebLogic 8.1 is now supported. See **Chapter 12, Management and Monitoring [591]**.
- Bugfixes
 - Fixed bug that could allow relations to get out of synch in the data cache when managed relations were enabled.

3.2.0 -- September 29, 2004

- New features
 - The reverse mapping tool interface in the workbench is now a guided wizard that allows the user to customize various aspects of the reverse mapping process.
 - The workbench will now try to compile java files into classes when the source is available but the class file is not.

- The reverse mapping tool can now generate inner classes for application identity classes with the `innerAppId` option. See [Section 7.2, “Reverse Mapping” \[495\]](#).
- Added an optional `DiagnosticContext` to Kodo's logging implementation. If set, all log lines from the configured `PersistenceManagerFactory` will be prefixed with the token. See [Section 3.2, “Kodo Logging” \[418\]](#).
- Added support for single-string JDOQL queries, a proposed JDO 2 feature. See [Section 11.9, “Single-String JDOQL” \[248\]](#).
- Added support for implicit JDOQL parameters and variables, a proposed JDO 2 feature. See [Section 11.3, “Advanced Object Filtering” \[236\]](#).
- Added preview of JDO 2 named query support. See [Section 11.10, “Named Queries” \[250\]](#) and [Section 17.4, “Named SQL Queries” \[324\]](#).
- Support for subqueries in JDOQL. See [Section 9.6.4, “JDOQL Subqueries” \[553\]](#).
- Added optional automatic management of inverse relations. See [Section 5.4, “Managed Inverses” \[458\]](#).
- Added `KodoPersistenceManager.checkConsistency` to check the consistency of the persistence manager cache without enlisting additional database resources. JDO 2 preview feature.
- Added new sample models to the `samples/` directory.
- Added `ExceptionAction` connection pool property to determine what to do when connections that have thrown exceptions are returned to the pool. Previous versions of Kodo always destroyed these connections, and that is still the default action. See [Section 4.1, “Using the Kodo DataSource” \[423\]](#).
- Many improvements to the workbench, including the ability to preview the SQL DDL for classes in the workbench, the ability to print components, single-click workbench starting, the ability to dynamically edit the classpath, and many user interface tweaks.
- Added ability to auto-externalize constant simple values (primitives, primitive wrappers and Strings) through metadata extensions. See [Section 5.5.5.1, “External Values” \[468\]](#).
- You can now configure Kodo to detach objects with their currently-loaded fields or to detach all fields rather than detaching based on the current fetch groups. See [Section 11.1.3, “Defining the Detached Object Graph” \[578\]](#).
- Added ability to automatically detach objects when the persistence manager closes, or when they are serialized. See [Section 11.1.4, “Automatic Detachment” \[580\]](#).
- Vertically-mapped inheritance hierarchies no longer require a class indicator column. See [Section 15.9, “Discriminator” \[281\]](#).
- You can now configure Kodo to outer-join to vertically mapped subclass tables when fetching data, on either a global or class-by-class basis. See [Section 5.7, “Eager Fetching” \[473\]](#).
- Improved eager fetching. Multiple relation fields of the same type can now be eager-fetched at once. Eager fetching using parallel selects now respects large result set settings, such that the related objects are selected for each "page" of objects brought into memory. Ability to specify that certain collection fields should be eager-fetched with joins when possible, rather than with parallel selects. See [Section 5.7, “Eager Fetching” \[473\]](#).
- Custom JDOQL extension methods can now take multiple arguments.
- Kodo now supports InterSystems Cache. See [Section 4.9, “InterSystems Cache” \[632\]](#).
- Added support for the `javax.jdo.query.SQL` query language introduced in the JDO 2 early draft specification. The `kodo.jdbc.SQL` query language is considered deprecated. See [Chapter 17, *SQL Queries* \[320\]](#) for details.

- SQL queries now support projections and custom result classes. See [Section 17.3, “SQL Projections” \[322\]](#).
- Support for grouping and having clauses in JDOQL queries, matching JDO 2 early draft specification. See [Section 11.7, “Aggregates” \[244\]](#).
- Support for setting result ranges on queries, matching JDO 2 early draft specification. Eager fetching works intelligently with query ranges so that the range is not affected by eager fetching and only eager data for the requested range is selected. See [Section 11.5, “Limits and Ordering” \[240\]](#).
- As per the JDO 2 early draft specification, JDOQL now supports the following new operators and functions: `instanceof`, `String.substring`, `String.indexOf`, `Math.abs`, `Math.sqrt`, `JDOHelper.getObjectId`. See [Section 11.2, “JDOQL” \[234\]](#).

Additionally, many JDOQL functions previously supported as Kodo query extensions are now official parts of JDOQL: `String.toLowerCase`, `String.toUpperCase`, `String.matches`, `Map.containsKey`, `Map.containsValue`.

- Allow access to public static fields in JDOQL, matching JDO 2 early draft specification. See [Section 11.2, “JDOQL” \[234\]](#).
- Allow single-quoted string literals in query filters, matching JDO 2 early draft specification. See [Section 11.2, “JDOQL” \[234\]](#).
- Support for distinct keyword in query result string, matching JDO 2 early draft specification. See [Section 11.6, “Projections” \[241\]](#).
- Default query result string to `distinct this as C`, where C is the unqualified candidate class name. Matches JDO 2 early draft specification. See [Section 11.8, “Result Class” \[246\]](#).
- Support for setting public fields of query result classes in addition to setter methods, matching JDO 2 early draft specification. See [Section 11.8, “Result Class” \[246\]](#).
- Added `Query.Extensions` map introduced in JDO 2 early draft specification.
- Queries executed through the JDO query facilities are now logged to the `kodo.Query` channel. See [Chapter 3, Logging \[417\]](#).
- Usage statistics for query results and collection and map type relations can now be viewed in the profiling tool. See [Chapter 13, Profiling \[603\]](#).
- Notable Changes
 - The `KodoPersistenceManager.flush` method now returns silently if no transaction is active, rather than throwing an exception. Complies with the JDO 2 early draft specification.
 - Removed the `kodo.jdbc.VerticalQueryMode` configuration property and associated runtime APIs. Also removed the `kodo.jdbc.JoinSubclasses` property and metadata extension introduced in 3.2.0b1. Use the new consolidated `kodo.SubclassFetchMode` property. See [Section 5.7, “Eager Fetching” \[473\]](#).
 - Upgraded Apache Commons Collections and Apache Commons Pool versions that ship with Kodo.
 - Kodo now includes the release candidate of the JDO 1.0.2 jar (`jdo-1.0.2.jar`) instead of the JDO 1.0.1 jar. This fixes an internationalization bug in the JDO 1.0.1 jar. It is no longer necessary to use the `i18nhelper Websphere_patch.jar` patch.
 - The JDO jar (now `jdo-1.0.2.jar`) file is no longer included in the resource adapter file (`kodo.rar`). The JDO jar file should be deployed in the global classpath for the application server.
 - Changed `MaxCharactersInAutoIncrement` property of `DBDictionary` to `MaxAutoIncrement`

NameLength to match naming conventions of other length restriction properties.

- `kodo.datacache.TangosolCache` now uses the `CacheFactory.getCache()` method if the `TangosolCacheType` property is left unset. As a result, the return value of `TangosolCache.getDistributedCache()` has been deprecated, and will not return accurate information if the `TangosolCacheType` configuration property is not set.
- Optimistic transactions in which no changes have taken place will no longer obtain and commit a datastore connection when the JDO transaction is committed.
- Simplified the **DBDictionary** class for easier extension and greater configurability. The changes may break existing custom dictionaries. The source code for the new dictionaries is included in your Kodo distribution to help you migrate your custom dictionaries.
- The query improvements in this release required changes to the **FilterListener** interface, and its `JDBCFilterListener` subclass. The source code for Kodo's built-in listeners is included in your Kodo distribution to help you migrate your custom listeners.
- The query improvements in this release also required minor changes to some **FieldMappings**. See the Javadoc and the samples in `samples/jdo/ormapping` for details.
- `kodo.jdbc.SQL` queries have been deprecated. Use JDO 2 preview SQL queries instead. See **Chapter 17, SQL Queries [320]**.
- The `FetchConfiguration.EAGER_FETCH_*` constants have been deprecated in favor of constants that more accurately reflect the semantics of each fetch mode. See **Section 5.7, "Eager Fetching" [473]**.
- Attempting to query on an interface or abstract class without any persistent implementors will throw an exception.
- Configuration of the management and profiling capabilities has been simplified. For more information, please see **Chapter 12, Management and Monitoring [591]** and **Chapter 13, Profiling [603]**.
- Bugfixes
 - Query parameter validation now ensures that you do not pass extra parameters to a `Query.execute()` invocation. This may cause some of your previously-functioning queries to fail to execute.
 - Corrected Empress database dictionary to use `TOLOWER` and `TOUPPER` SQL functions. Bug 964.
 - Removed restriction limiting arguments to the `startsWith` and `endsWith` JDOQL methods to literals and parameters. Bug 970.
 - Fixed possible exception when attaching new embedded objects.
 - Worked around JDO library bug with potential infinite recursion when printing out stack traces that contain a failed object whose `toString` method accesses persistent fields. Bug 979.
 - Fixed exception when supplying a null value for an implicit parameter.
 - Query caching behaves properly with mutable parameter types (`Collection`, `Date`) that are changed between executions (bug 959)
 - Queries that return no results are properly cached.
 - Projections can now contain the same column multiple times; bug 853.
 - Added support for non-persistence capable query variables; bug 720.
 - All return types from projections and aggregates, whether executed in-memory or in the data store, now exactly match the

types specified by the JDO 2 early draft specification. Bug 721.

- Fixed some cases of not being able to use variables in projection and aggregate queries executed in-memory. Bug 713.
- Fixed bug that prevented a lock-group of none from working in some cases.
- Fixed a bug that prevented the reverse mapping tool from recognizing the -s cmd-line argument shortcut for the -schemas option.

3.1.5 -- August 11, 2004

- Notable Changes
 - When setting `kodo.ProfilingInterface` to `export`, default export interval is now -1, indicating that only a final export will be created.
- Bugfixes
 - Fixed possible exception when attaching a new embedded object.
 - Included check for lock-group when using state-image version indicator.
 - Final profiling export when setting `kodo.ProfilingInterface` to `export` is now produced.
 - Fixed `DBDictionary` to allow column auto-increment sequence names longer than 31 by adding `MaxCharactersInAutoIncrement` property.

3.1.4 -- July 9, 2004

- New features
 - `ClassDBSequenceFactory` can now ignore horizontally mapped classes when determining primary key values.
 - Firebird is now a supported database. See **Section 4.7, “Firebird” [631]**.
 - Borland Interbase is now a supported database. See **Section 4.2, “Borland Interbase” [629]**.
- Bugfixes
 - Corrected bug in `appidtool` that could result in invalid application identity subclass generation when the application identity superclass had one or more `Date` primary key fields.
 - The reverse mapping tool now honors the `DBDictionary`'s `UseSchemaName` property when deciding whether to qualify table names in generated mappings.
 - Fixed recently-introduced `NullPointerException` in some one-many mappings.

3.1.3 -- June 21, 2004

- New features

- Subclasses in an application inheritance hierarchy can now define additional primary key fields. See **Section 4.5.2.1, “Application Identity Hierarchies” [190]**.
- The Kodo Development Workbench now includes options for previewing the metadata and mappings in XML form. In addition, you can now edit foreign keys and use them in the visualization process.
- The reverse mapping tool accepts a new option `-- -blobAsObject/-bo --` to map binary columns to `java.lang.Object` fields, rather than to `byte[]` fields. Enabling the option mirrors the tool's behavior in Kodo versions prior to 3.1.2.
- Kodo will now issue warnings to the log when a mapping contains unrecognized attributes, or when a metadata extension contains an unrecognized key. The log message will include suggestions for similarly name attributes.
- Kodo will now issue warnings to the log when an unrecognized property name is specified in the `kodo.properties` configuration file. The log message will include suggestions for similarly named properties.
- Notable Changes
 - The `AbstractStoreManager.newInstance` method no longer exists; use the `KodoStateManager.initialize(Class, JDOState)` method in its place.
- Bugfixes
 - Fixed bug with id class validation problems with NetBeans plugin.
 - Fixed bug that could cause errors after rollback of a transaction in which an object with dependent relations was deleted.
 - Fixed externalization bug that could lead to a `ClassCastException` when one container type was externalized to another container type.
 - Fixed bug with caching of queries that use empty `Collection` parameters (bug 944).
 - Fixed bug in which reverse mapping tool generated invalid Java code for binary columns.
 - Changed DB2 SQL generation to not use `CROSS JOIN`, which DB2 does not understand.
 - Fixed bug that prevented Workbench from allowing you to edit metadata for field that are non-persistent by default.
 - Fixed query bug that could result in a `NullPointerException` when attempting to constrain a variable to a collection or map passed in as a parameter (or traversed from a persistence-capable parameter).
 - Fixed mapping bug when mapping an embedded field of a horizontally-mapped superclass to a subclass table.
 - Corrected an unnecessary `SELECT DISTINCT JDOCLASS` when using `PersistentClasses` list.

3.1.2 -- May 21, 2004

- Bugfixes
 - Fixed 3.1.1 bug that prevented use under some licenses.

3.1.1 -- May 20, 2004

- New features

- Support for embedded one-to-one mappings in Kodo Development Workbench.
- Support for data compression and filtering when using remote persistence managers. See [Section 11.2.4, “Data Compression and Filtering” \[586\]](#).
- Support for byte array primary key fields for legacy schemas that use binary columns for primary keys.
- XML Store Manager sample now included.
- The Kodo workbench now includes a live JDOQL query executor and result browser.
- Notable Changes
 - Changed default storage directory for Kodo Development Workbench from the current working directory to `${user.home}/solarmetric`. Kodo Development Workbench will handle the migration for you. However, you can choose to override this behavior by using the `-s` argument to specify a different location.
 - Byte array fields are now mapped using the byte array field mapping by default, rather than the blob field mapping. The byte array mapping does not serialize its data, while the blob mapping does. This should not affect existing mappings.
 - The reverse mapping tool now maps binary columns to `byte[]` fields, rather than Object fields.
 - Downgraded non-locking SELECT log messages from WARN to INFO.
 - Integration of `remotejmxtool` with unsupported application servers can now be plugged in. See [Section 12.2.1, “Remote Connection” \[595\]](#) for details.
 - Changed `kodo.runtime.LockManager` plugin API to allow the lock manager to access the connection info we are loading from, if any. This is useful for plugins that implement the `kodo.jdbc.runtime.JDBCLOCKManager` interface.
- Bugfixes
 - Changed class loading in Kodo tools so that static initializers of user-defined classes are not invoked in dev-time operation.
 - Fixed a possible constraint violation when using maps with restrict-action foreign keys to persistent object values. The error occurred when changing an existing map key and deleting the old value object.
 - Fixed a bug that could result in mapping errors when using numeric constant joins in combination with the dynamic schema factory.
 - Fixed a bug that could result in mappings to be generated for the implicit "jdoDetachedObjectId" field that is created when enhancing a detachable class. By default, any field starting with "jdo" will no longer be considered persistent unless it is explicitly declared.
 - Corrected an inefficiency in object locking that could result in a SELECT FOR UPDATE being issued for an already-locked object.
 - Fixed a bug that sometimes prevented Kodo from obtaining a requested database lock if the object was found in the data cache.
 - Removed potential deadlock in `JDBCConfiguration`.
 - Fixed bug that caused spurious graphical glitches and exceptions in JMX Management Console.
 - Removed a limitation that prevented non-serializable persistent instances from being used as parameters in a remote query. Note that this limitation is still in place for non-persistent instances.

- Fixed bug that caused certain custom field mappings using `java.sql.Date/Timestamp` to get a `ClassCastException`.
- Enabled App Id Tool to generate object identity classes for horizontally mapped classes.
- Fixed `ColumnVersionIndicator` to throw correct lock exception on certain conditions with deleted rows.

3.1.0 -- April 23, 2004

- New features
 - Added new configuration properties and runtime APIs for enhanced control over object locking. See **Section 9.4, “Object Locking” [542]** for details.
 - Kodo Development Workbench includes a number of new features and changes, including the ability to edit version and class indicators in the visualization editor.
 - You can now configure Kodo to evict objects from the data cache whenever you evict them from a persistence manager. See **Example 10.9, “Automatic Data Cache Eviction” [565]**.
 - Added methods to `KodoHelper` to retrieve the data cache for an object or class.
 - Added example XML based `AbstractStoreManager` called `kodo.xmlstore.XMLStoreManager`.
 - Added support for large result set handling for fields declared as type `java.util.Set`.
 - Added support for constant joins in many-to-many relations, including mapping both a one-to-one and a many-to-many into the same join table.
- Notable Changes
 - Object locking APIs have changed. See **Section 9.4, “Object Locking” [542]** for details.
 - Added support for interface mappings in JBuilder and SunONE/NetBeans plugins in addition to other usability and stability fixes. Users of these plugins should install the new versions.
 - `ClassDBSequenceFactory` now has a `main (String[] args)` method to create and drop required schema components.
 - The `DBDictionary` now supports the `DefaultSchemaName` parameter. See **Section 4.4, “Database Support” [428]**.
 - Optimized statement batching within groups of SQL updates ordered to meet foreign key constraints.
 - The `StoreManager` API has changed slightly in light of the object locking enhancements in this release. See its **Javadoc** for details.
- Bugfixes
 - Fixed a bug in which the new `lockPersistent` APIs could cause errors if used during an optimistic transaction.
 - Fixed SQL generated for removing columns on Empress.
 - Created special case for LOB handling when using a Weblogic datasource connecting to an Oracle database to circumvent the fact that Weblogic wraps the native Oracle LOB-handling classes.
 - Fixed bug in which passing duplicate oids to `KodoPersistenceManager.getObjectsById` could result in internal errors.

3.1.0 RC2 -- March 24, 2004

- New features
 - Added the ability to use remote persistence managers over HTTP/HTTPS. See **Chapter 11, Remote and Offline Operation [577]** for details.
 - Added an abstract store manager, which is a building block for allowing you to add support for non-relational data stores that Kodo does not support. See the **kodo.abstractstore javadocs** for documentation.
 - Added a **SequenceGenerator.ensureCapacity(int count)** method. Invoke this method to provide a hint to a sequence generator about how many times its **next()** method will be invoked.
 - Added support for Empress database.
- Notable Changes
 - Kodo Development Workbench has added some usability improvements in addition to a variety of bugfixes.
 - Eclipse plugin has now been changed to 2.1.0. This new version includes the ability to enhance and run Mapping Tool on multiple .jdo files at once. In addition, a more full range of Mapping Tool options such as readSchema and ignoreErrors are now available. The enhancer builder now delays til the end of building to avoid re-processing of metadata (and potential classloader issues related to this).

This version of the plugin includes also a number of bug fixes. It is recommended that you uninstall the old version of the plugin by completely removing the old kodo.eclipse_2.0.0 directory and installing the new one.
 - Deprecated **SequenceGenerator.getNext()** in favor of **SequenceGenerator.next()**, which returns an unboxed long rather than a **java.lang.Number**.
 - Management capability now disabled by default. To find/create an MBeanServer and register MBeans set the **kodo.MBeanServerStrategy** configuration property. See **Chapter 12, Management and Monitoring [591]** for details.
- Bugfixes
 - Fixed bug that caused intermittent errors in remote persistence managers when the primary key values of an application identity instance were changed after it was made persistent, but before commit.
 - Improved JBuilder plugin stability. Bugs fixed include proper handling of cancel, drop, and file organization.

3.1.0 RC1 -- March 10, 2004

- New features
 - Kodo now uses its own logging framework by default instead of the Commons Logging framework. To use the Commons Logging framework as in older versions of Kodo, set the **kodo.Log** property to **commons**. See **Chapter 3, Logging [417]** for details.
 - Kodo Workbench now includes the ability to dynamically edit from the Visualization editor. Mapping Tool actions now work on the current instead of the saved versions of mapping information.
 - The **kodo.jdbc.SynchronizeMappings** property (which was known as **com.solarmetric.kodo.impl.jdbc.SynchronizeSchema** in version of Kodo prior to 3.0) now allows Kodo to dynamically attempt to build O/R mappings and the database schema at runtime. See **Section 7.1.4, "Runtime For-**

ward Mapping” [495].

- Notable Changes
 - Kodo's JCA connection factory can now be safely cast to a `KodoPersistenceManagerFactory` rather than just a `PersistenceManagerFactory`.
 - The reverse mapping tool now automatically strips illegal characters from table and column names when mapping them to Java identifiers. This is mostly a bug fix, but also affects users who have reverse customizer properties files renaming fields for which Kodo used to include invalid characters.
 - If the `DBDictionary.StoreCharsAsNumbers` property is set to `false`, `CHAR(1)` columns will be reverse-mapped to Java char fields rather than strings.
- Bugfixes
 - Fixed bug that caused Kodo to sometimes return an inaccurate size for query results when large result sets were enabled, the query involved to-many joins, and Kodo was configured to use `SELECT COUNT` to calculate the size.
 - Fixed bug preventing objects with null relations from being returned when those relations were eager-fetched and were to subclasses using vertical inheritance.
 - Fixed bug that caused a parse error when attempting to case with the full class name in a JDOQL filter string (i.e. `"((com.xyz.Foo) foo).bar == x"`). Bug 869.
 - Fixed bug in which fields declared in the class of the cast were sometimes not recognized when casting in JDOQL filter string. Bug 805.
 - The `SELECT COUNT` subselect issued when testing for a null 1-1 or empty collection/map in a query filter now uses the fully qualified table name (including schema).
 - Fixed `NullPointerException` when passing an array or collection containing null elements to certain persistence manager methods such as `retrieveAll`.
 - Fixed bug involving the customization of reverse-mapped field names. Bug 881.
 - Fixed reverse-mapping problem with Informix databases.
 - Fixed error when you add a new instance to an ordered one-many relation, flush, then extensively modify or re-order the elements of the relation, then commit. Bug 887.

3.1.0b1 -- February 16, 2004

- New features
 - Kodo now supports the "horizontal" O/R inheritance model, where each leaf of the inheritance hierarchy is mapped to a separate table. for details.
 - Added the ability to use Kodo from remote client machines communicating with a persistence manager server. See **Section 11.2, “Remote Managers” [581]** for details.
 - Improved efficiency of the query compilation cache. After a query is compiled or executed once, subsequent compilations or executions of queries with the same properties (even in different persistence managers) are much faster.
 - Added custom lock group support, allowing field-level optimistic locking granularity. See **Section 5.8, “Lock Groups” [476]** for details.

- Added object locking APIs to the `KodoPersistenceManager` for fine-grained control over object locking. You can also plug in your own locking scheme through the `kodo.LockManager` configuration property.
- Added a `cancelAll` method to the `KodoPersistenceManager` that can be used to cancel any outstanding database statements issued by a `PersistenceManager`.
- Added a new option for controlling how Kodo queries against class hierarchies that use vertical inheritance mapping. See [Section 5.7, “Eager Fetching” \[473\]](#) for details.
- Significantly improved technology preview of Kodo Management / Monitoring capability based on JMX. Includes local and remote management functions, management of Data cache, Prepared Statement cache, and Kodo Datasource Connection Pool, and advanced performance analysis. See [Chapter 12, Management and Monitoring \[591\]](#) for details.
- Notable Changes
 - `kodo.jdbc.meta.ClassMapping` and `kodo.jdbc.meta.FieldMapping` are now interfaces. Custom implementations that previously extended these classes directly should now extend `kodo.jdbc.meta.AbstractClassMapping` and `kodo.jdbc.meta.AbstractFieldMapping`.
 - The `KodoQuery.FLUSH_*` constants have been deprecated in favor of equivalent constants in the `FetchConfiguration` interface for easier access.
- Bugfixes
 - Fixed bug with invoking `size()` on query results returned from SQL queries when using lazy result support.

3.0.3 -- February 20, 2004

- New features
 - Added a `ref-constant` attribute type to mapping data, so that you can perform constant joins (see [Section 7.6, “Non-Standard Joins” \[506\]](#)) on reference foreign keys, such as those used in one-many mappings.
 - Manually flushing the persistence manager before commit now releases all hard references to flushed dirty objects, allowing you to insert or update an unlimited number of objects within the same transaction without running out of memory.
 - Added the new `kodo.runtime.PreDetachCallback`, `kodo.runtime.PostDetachCallback`, `kodo.runtime.PostAttachCallback` and `kodo.runtime.PreAttachCallback` interfaces that allow instances to be notified when they are being detached or attached.
 - Added support for simulating auto-increment fields under Oracle by using triggers. See [Section 4.4.3, “OracleDictionary Properties” \[435\]](#).

Notable Changes

- Deprecated the `KodoPersistenceManager.getState` method in favor of the new `KodoPersistenceManager.getStateManager` method, which takes in a persistence capable instance rather than an object id. The actual instance is required to make sure that the correct state is returned when multiple persistent-new application identity objects have the same object id. This is possible if they are persisted before their primary key fields are set to unique values and the transaction has not yet committed.
- The data cache now does not do any copying of `Locale` objects, as they are final and immutable. This means that if you cache an object with a `Locale` reference and then load that object from cache at a later time, the `Locale` will be identical (`==` will pass) in both objects.
- The `Mappings.getForeignKey` and `Mappings.setForeignKey` custom mapping helper methods now auto-

matically suffix the given attribute prefix with `-column` (use `null` for an attribute of `column`).

- Bugfixes
 - Fixed bug that sometimes prevented changes in positions of ordered list elements from being committed to the database.
 - Stopped Kodo from occasionally iterating large result set fields on load and flush.
 - Fixed bug that left open result sets when using **large result set fields** of first class objects.
 - Fixed bug with caching of `Date` and mutable (proxied) custom SCO field types.
 - Fixed bug with improper lookup of unloaded related objects from cache (bug 836).
 - Fixed bug that caused eagerly-fetched to-many fields to sometimes not contain all elements if the field elements were constrained in the query being executed (bug 855).
 - Fixed bug that could cause exponential rise in commit time when committing many interrelated new objects.
 - Fixed bug that prevented multiple new instances from being able to be attached if they used application identity (bug 848).
 - Fixed bug that prevented attaching a graph that made a newly added instance persistent if it was reachable via multiple paths (bug 860).

3.0.2 -- January 24, 2004

- New features
 - Added a `TableTypes` property to the `DBDictionary` to allow configuration of the table types that will be considered when reflecting on the schema.
 - Added a `jdbc-version-ind-indexed` and `jdbc-class-ind-indexed` class metadata extensions to control the indexing of version and class indicator columns, respectively.
 - The `DBDictionary` now supports the `InitializationSQL`, `CatalogSeparator`, and `UseSchemaName` parameters. See **Section 4.4, “Database Support” [428]**.
 - Allow application identity classes to use custom sequence factories for their sequence generators.
- Notable Changes
 - Made version indicator columns indexed by default. The next time you run the mapping tool's `refresh` action, you may see indexes added to existing version indicator columns. To prevent this, set the new `jdbc-version-ind-indexed` class metadata extension to `false`.
 - Fixed bug that prevented the mapping tool's `buildSchema` action from adding indexes to the class and version indicator columns.
 - Changed the default `BatchLimit` `DBDictionary` setting for Oracle 9.2 Driver to handle statement batching issues. Users connecting using the recommended 9.0.1 driver are unaffected.
- Bugfixes
 - Fixed bug that caused invalid SQL when using the mapping tool's `buildSchema` action under Sybase.

- Fixed bug that prevented the mapping tool's `buildSchema` action from adding indexes to the class and version indicator columns.
- Fixed bug that caused invalid SQL when using large result set collection or map fields with types with compound primary keys.
- Fixed bug that prevented inserts or updates of BLOBs over 4K in Oracle if the BLOB column had a NOT NULL constraint.
- Fixed rare `NullPointerException` in query compilation cache.
- Fixed problems with `SybaseDictionary`'s handling of `BigDecimal` and `BigInteger` values.
- Fixed bug that sometimes led to duplicate objects in query results due to missing `DISTINCT` in database select.
- Fixed issue with re-attaching detached instances with generic/unknown type fields.
- Fixed metadata parsing issue with classes loaded under the bootstrap classloader in certain situations.
- Fixed memory leak when invoking `Query.close()` (as opposed to `Query.closeAll()`) on a cached query.

3.0.1 -- December 16, 2003

- New features
 - Includes technology preview of the standalone Kodo Development Workbench. Kodo Development Workbench provides integrated mapping tools for your Kodo development, including metadata editors, visual relational graph analysis, configuration wizards, and access to development tools such as **SchemaTool** and **Reverse Mapping Tool**
 - Added new Byte Array Field Mapping. This mapping avoids serialization of byte array fields for interactions with non-Java applications.
 - The `sqlline.jar` utility is now included in the Kodo distribution. It is useful for a unified command interface for any database. See **Section 4.15, “The SQLLine Utility” [449]**, and for complete documentation, see **<http://sqlline.sourceforge.net>**.
 - Added support for expressing nested O/R mapping extensions via XDoclet.
 - Introduced a cache for shareable query-related information. This improves query compilation times in many situations. See **Section 2.7.52, “kodo.QueryCompilationCache” [409]** for details.
 - Added a `class-criteria` attribute to the one-one mapping.
 - Added support for naming and configuring multiple data caches via the `kodo.DataCache` configuration property. See **Section 10.1.1, “Data Cache Configuration” [561]** for details.
- Notable Changes
 - Query extensions and aggregate queries are now included as part of Kodo Standard Edition; Enterprise Edition is no longer required to utilize these features.
 - Changed connection pool to reduce concurrency when creating and closing connections.
 - Changed the default value of the `kodo.AutoClear` flag to `all` to comply with JDO 1.0.1 spec section 5.8. Note that this gives behavior similar to previous versions of Kodo; the change we made to the default behavior in Kodo 3.0 was incorrect.

- Connection Decorators and JDBC listeners now do their work on all connections, inclusive of DBDictionary access.
- Changed the handling of dependent fields to allow you to move dependent objects to other fields in a transaction. Kodo now only deletes dependent objects that have been removed from their owning field or had their owning object deleted, and that have not been assigned to any other field of any object. This analysis occurs on flush.
- Bugfixes
 - Fixed bug that prevented runtime licenses from working.
 - Optimized query compilation for datastore execution, not in-memory execution.
 - Fixed bug that inserted invalid rows into map tables when an existing key of a persistent map field was given a new value.
 - Fixed bug that caused direct SQL queries to throw an exception when executed in a transaction, even when no objects had been modified in the transaction and/or the global `javax.jdo.IgnoreCache` property was set to `true`.
 - Fixed bug that prevented the file mapping factory from working correctly when the path to metadata files contained spaces.
 - Fixed bug with Ant and Mapping Tool classloader conflict which caused `ClassNotFoundException`s.
 - Fixed bug 798 -- potential memory leak when query caching is enabled.
 - Fixed a bug that caused inverse-based one-many and one-one mappings without an inverse-owner to sometimes produce bad SQL if the inverse columns were mapped to other fields as well, or were not nullable.

3.0.0 -- November 7, 2003

- New features
 - Kodo now supports direct SQL queries and stored procedure calls through the `javax.jdo.Query` interface. See **Chapter 17, *SQL Queries* [320]** for details.
 - Technology preview of Kodo Management / Monitoring capability. See **Chapter 12, *Management and Monitoring* [591]** for details.
- Notable Changes
 - Added documentation for deploying in JRun 4.
 - The `ext:namedQuery` JDOQL extension has been replaced with the `kodo.MethodQL` query language. See **Section 9.6.5, “*MethodQL*” [554]** for details.
 - By default, the mapping tool no longer reads the entire existing schema on startup, though this option is still available. Also, the mapping tool does not examine or manipulate indexes, foreign keys, or primary keys on existing tables by default, though these options, too, are available as flags.
- Bugfixes
 - Fixed a bug that produced incorrect and sometimes invalid SQL when eager-fetching an inverse one-one relation.
 - Fixed a bug introduced in RC 3 that could cause collections and maps to be loaded as null whenever the data cache was enabled.
 - Fixes bugs in the IDE plugins. Please re-install any previous installations before installing the new versions of the plugin.

3.0.0 RC4 -- October 31, 2003

- Notable Changes
 - The `jdbc-use-*` metadata extensions have been renamed to `jdbc-*-name` extensions. So, for example, the `jdbc-use-class-map` extension is now `jdbc-class-map-name`. This change actually happened in RC3, but was not fully documented.

3.0.0 RC3 -- October 31, 2003

- New features
 - Added a `KodoHelper.getSequenceGenerator` method to ease obtaining a sequence for application identity classes.
 - Built-in support for Borland JDataStore, as well as Microsoft Access and Microsoft Visual FoxPro (using a JDBC-ODBC server bridge like DataDirect, but not the Sun JDBC-ODBC bridge).
- Bugfixes
 - Fixed synchronization problem that could result in a `ConcurrentModificationException` when Kodo is used under high load with managed transactions.
 - Fixed problem with incremental flushing that made it impossible to edit an existing object, flush, and then delete.

Notable changes

- The behavior of the `data-cache-timeout` metadata extension has changed considerably. In previous release candidates, a value of 0 meant that a class should never expire, and a value of -1 meant that the class should be excluded from the cache altogether. As of this version, a value of -1 means that the data in the cache should not expire, and is the default. 0 is no longer a valid value.

Additionally, the `data-cache-name` metadata extension's name has been changed to `data-cache`. Its role has been expanded to control disabling caching for a particular cache. To disable caching, set the `data-cache` extension to `false`. The default value for this extension is `true`.

So, to sum up, if you had a `data-cache-timeout` extension set to 0, you will get an exception when the metadata is parsed. Change the value to -1 instead. If you had set the extension to -1, then things are a bit trickier -- this will change the semantic behavior of your class unless you remove the extension and set the `data-cache` extension to `false`.

- Removed the `xa` option from the `kodo.TransactionMode` configuration property. When using an XA data source or other data source that is automatically involved in the global transaction, set `kodo.TransactionMode` to `managed` and set the new `kodo.jdbc.DataSourceMode` property to `enlisted`.
- The default value for `javax.jdo.option.IgnoreCache` has been changed from `false` to `true`.
- Refactored data caching to not lock the cache as a whole when loading data from it or storing data into it. This change improves the concurrency of the cache.
- Removed the `kodo.DataCacheConnects` property, as it is not needed now that locks are not obtained on the data cache.
- Re-worked SunONE/NetBeans and JBuilder plugins to remove a number of bugs as well as to improve the UI. Editors now incorporate commonly used Kodo extensions. For users of earlier versions, we recommend un-installing and re-installing the plugin.

- Made assorted minor tweaks to prepared statement and query caching.

3.0.0 RC2 -- October 9, 2003

- New features
 - None
- Bugfixes
 - Fixed an optimistic locking error when the data cache is enabled.
 - Fixed some minor attach/detach bugs.
 - Fixed a bug in which persistent non-transactional objects were not cleared and reloaded when dirtied in an optimistic transaction. This could lead to false optimistic lock exceptions. See notable changes section below for details.
- Notable changes
 - Added the `kodo.DataCacheConnects` property to determine whether the data cache obtains a connection before each cache access. See the last list item in **Section 10.1.8, “Known Issues and Limitations” [575]** for details.
 - Kodo now clears and reloads persistent non-transactional objects when they are dirtied in an optimistic transaction. This is correct behavior as far as the spec is concerned, but can result in lower performance than the previous non-clearing behavior under certain usage patterns. Also, this change means that non-transactional writes can no longer be committed. Users who would like to continue with the old non-clearing behavior in order to increase performance or allow non-transactional writes to be committed should set the `kodo.AutoClear` configuration property to `all`. Users of optimistic transactions and non-transactional reads who choose not to set this property should watch out for the following usage pattern:

```
Person p = (Person) pm.getObjectById (oid, true);
pm.currentTransaction ().begin ();
p.setName ("New Name"); // this now causes a reload of the object state!
pm.currentTransaction ().commit ();
```

If you are looking up data in order to modify it, make sure to look it up within the transaction rather than just before the transaction, and thus avoid a state refresh.

3.0.0RC1 -- 23 September 2003

- New features
 - Support for **detaching and attaching** instances from a persistence manager, allowing applications to more easily use a "data transfer object" pattern.
 - Support for collection and map fields that are **backed by large result sets**.
 - Support for additional settings to control the handling of **large result sets**.
 - Better exception messages when mappings can't be found or fail validations against the schema.

- Bugfixes
 - Fixed many eager-fetching SQL errors.
- Notable changes
 - Kodo now uses non-scrolling result sets by default, and fetches all query results up-front by default. See the **large result set** section of the reference guide for how to configure large result set handling if needed.
 - The `JDBCQuery`'s `JoinSyntax` property has been moved into the query's `JDBCFetchConfiguration`. You should no longer cast to `JDBCQuery`, since that cast can fail when the data cache is enabled. Use the fetch configuration instead.

3.0.0b2 -- 3 September 2003

- New features
 - Expanded **smart proxies** to include change-tracking for lists.
 - Kodo now examines the initial field value of managed fields and stores any custom comparators for use when loading data into that field from the database.
 - Added the `kodo.RestoreMutableValues` configuration property.
 - **IDE plugins** have been re-implemented for Kodo 3. Support for NetBeans, SunONE Studio, JBuilder, and Eclipse have been re-added. Uninstall previous versions of the plugin, and follow the documentation for each plugin's installation.
 - New **externalization** system for storage of field types not supported by JDO without resorting to serializing or requiring custom mappings. Replaces the old stringification mapping.
 - Support for aggregates and projections in queries. See **Chapter 11, Query [232]** for details.
 - Added a `matches` query extension for limited regular-expression matching in JDOQL. See the **Included Query Extensions** documentation for details.
 - Documentation for how to use the latest builds of XDoclet to generate JDO metadata from source comments is now in the Integration chapter of the reference guide. A new XDoclet sample was also added to the `samples/` directory.
 - Added APIs for `get/setRollbackOnly` to the `KodoPersistenceManager` interface.
- Bugfixes
 - Fixed a bug in the first beta that prevented MySQL tables with VARCHAR columns from being created correctly.
 - Fixed a bug in the first beta that prevented eager-fetching from working efficiently. Also fixed some cases that could lead to stack overflow errors when using eager fetching.
 - Fixed a case in which compound primary keys could sometimes cause array index out of bounds errors when retrieving objects.
 - Fixed the Kodo 2 migrator tool, which sometimes specified arrays with the `<collection>` element in the migrated metadata.
 - Improved support for columns shared by both relation foreign keys and simple value fields.
 - Improved error messages when setting the same column to multiple different values or when trying to insert multiple objects with the same oid.

- Notable changes
 - Configuration properties specifying system plugins have been consolidated. See the **Plugin Configuration** section of the **Configuration** chapter for details. Beta 1 users may want to re-run the Kodo 2 properties migration tool on their old Kodo 2 properties instead of modifying their beta 1 properties by hand.
 - The default **mapping factory** has been changed to the file-based factory. If you were using the default database mapping factory in beta 1, either **switch to the file mapping factory**, or add the following line to your properties file:

```
kodo.jdbc.MappingFactory: db
```

- Changed the default table and column names for the `DBSequenceFactory`. If you were using the DB sequence factory in beta 1, either re-run the Kodo 2 properties migration tool, or add the following line to your properties:

```
kodo.jdbc.SequenceFactory: PrimaryKeyColumn=PKX, SequenceColumn=SEQUENCEX, \  
  TableName=JDO_SEQUENCEX
```

- The stringification field mapping was replaced by the **externalization** framework.
- The `stringContains` and `wildcardMatch` query extensions have been deprecated in favor of the `matches` query extension.

3.0.0b1 -- July 24, 2003

- New features
 - New mapping system, providing more flexible mapping options. See the chapter on **Object-Relational Mapping** for more information.
 - Pluggable system for storing mapping information, with built-in options for storing mapping information in the database, in JDO metadata extensions, and in a separate mapping file. See the section on the **Mapping Factory** for more information.
 - Support for embedded 1-1 mappings, including nested embedded mappings with no limit on nesting depth. See the section on **Embedded One-to-One Mapping** for more information.
 - More complete mapping support for interfaces, including support for interfaces as the element type of collections, and the key and value types of maps. See the section on **Field Mapping** for more information.
 - Support for other-table mappings with outer joins. See the section on **Value Mapping** for more information.
 - Support for 1-many fields without an inverse 1-1 field. See the example **Using a One-Sided One-to-Many Mapping** in the section on **One-to-Many Mappings** for more information.
 - Support for first class objects as map keys. See the sections on **Many-to-N Map Mapping**, **Many-to-Many Map Mapping**, **PC Map Mapping**, **PC-to-N Map Mapping**, **PC-to-Many Map Mapping**, and **Many-to-PC Map Mapping** in the section on **Field Mapping** for more information.
 - Support for non-standard joins, including partial primary key joins, non-primary key joins, and joins using constant values. See the section on **Non-Standard Joins** for more information.

- New samples for custom field mappings. The `samples/jdo/ormapping` directory of the Kodo JDO distribution includes examples of custom mappings.
- Support for automatically ordering SQL operations to meet all foreign key constraints, including circular constraints.
- Support for `javax.jdo.option.NullCollection`.
- Support for timestamp and state-based optimistic lock versioning, and for custom versioning systems. See the section on Version Indicators for more information.
- More configuration options for connection pooling. See the section on **`kodo.ConnectionFactoryProperties`** for more information.
- Support for SQL logging on third-party `javax.jdo.DataSources`.
- Configurable **eager fetching** of 1-1, 1-many and many-many relations. Potentially reduces the number of database queries required when iterating through an extent or query result and accessing relation fields of each instance.
- Ability to obtain both managed and unmanaged persistence managers from the same `PersistenceManagerFactory`.
- Support for auto-increment fields.
- Better support for auto-incrementing primary keys.
- More optimized **SQL batching**.
- Fail-fast error messages when object-relational mappings, class definitions, and schema are not in synch.
- Automatic schema generation now names schema components better, and automatically avoids naming conflicts with existing schema components.
- Simplified package structure and plug-in APIs. See the section on **JDO Runtime Extensions** for more information.
- Bugfixes
 - Fields in the same class hierarchy which share the same name no longer cause an invalid schema.
- Notable changes
 - A series of steps must be followed in order to migrate from Kodo 2.4 or Kodo 2.5 to Kodo 3.0. Please see [Migrating from Kodo 2 to Kodo 3](#) for more information.
 - The `schematool` has been replaced with the more powerful `mappingtool`. The `schematool` still exists, but now has a different purpose. See the section on **Schema Tool** for more information.
 - In Kodo 2.x, the default-fetch-group was not loaded when an object transitioned from hollow to a stateful state because a non-default-fetch-group field was loaded. Because `InstanceCallbacks.jdoPostLoad()` is invoked after the default-fetch-group is loaded, this meant that the `jdoPostLoad()` callback was not invoked in some circumstances when it might otherwise be expected to be invoked. kodo 3 always loads the default-fetch-group when an object transitions from hollow, so `jdoPostLoad()` will now be invoked in situations in which it was not invoked in the past.
- Beta Notes
 - Integration with the supported IDEs is not included in 3.0.0b1. IDE integration will be included in later distributions.
 - XDoclet integration is not included in 3.0.0b1. It will be added in a later release.
 - Support for DB2, Informix and Sybase is not included in 3.0.0b1. Support for these databases will be included in later dis-

tributions.

- Enterprise integration is not fully tested in 3.0.0b1. Full testing and support for will be included in later distributions.

2.5.5 -- 18 October 2003

- Bugfixes
 - Fixed synchronization problem in `EEFactoryHelper` that could result in a `ConcurrentModificationException` when Kodo is used under high load with managed transactions.
 - Fixed problem with checking the optimistic lock version of a subclass that uses a vertical inheritance mapping strategy.
- Notable changes
 - Refactored data caching to not lock the cache as a whole when loading data from it or storing data into it. This change improves the concurrency of the cache.
 - Removed the `com.solarmetric.kodo.DataCacheConnects` property, as it is not needed now that locks are not obtained on the data cache.
 - Made assorted minor tweaks to prepared statement and query caching.

2.5.4 -- 7 October 2003

- Bugfixes
 - Fixed bug with extents not closing resources with certain `ResultList` implementations due to internal iterators not closing.
 - Fixed bug with data caching and incremental flushing and loading that could result in incorrect `OptimisticLockExceptions` being thrown.
 - Fixed bug with data caching that could result in deadlocks when used in conjunction with table-level or page-level locking.
- Notable changes
 - Added the `com.solarmetric.kodo.DataCacheConnects` property to determine whether the data cache obtains a connection before each cache access. Note that this property defaults to `false`, which mirrors Kodo 2.5.2 behavior. Users who experienced data cache hangs in Kodo 2.5.2 because of empty connection pools should set this property to `true` to mirror Kodo 2.5.3 behavior.

2.5.3 -- 27 August 2003

- Bugfixes
 - Fixed bug with invalid `SQLServer SQL92` generation when using pessimistic locking.
 - Addressed performance issues caused by recomputing persistent type lists and subclass lists too often.
 - Fixed result list implementation used by the query caching framework to properly lazily load results.

- Fixed `DataCacheStoreManager` to properly deal with creating a new query based on a template that is a `CacheAwareQuery`, and changed `CacheAwareQuery` to have a `writeReplace()` method that returns the delegate query object rather than the cache-aware query.
- Fixed bug that prevented custom query extensions from being recognized.
- Fixed bug with data caching and incremental flushing that could result in incorrect `OptimisticLockExceptions` being thrown.
- Changed on-demand `ConnectionRetainMode` to only obtain a single connection per `PersistenceManager`. In other words, if a PM is using a connection (for example, while iterating a large query result), and it performs an operation that requires a connection, it will use the previously-obtained connection rather than obtaining a new connection. This reduces resource consumption, and helps to avoid possible race conditions while obtaining connections. If the old on-demand `ConnectionRetainMode` is necessary for some reason, it can be activated by setting `com.solarmetric.kodo.impl.jdbc.ConnectionRetainMode` to `legacy-on-demand`.
- Fixed potential race condition when performing operations on the data cache that might require a trip to the data store.
- Notable changes
 - Improved validation of application ID object-id classes may cause errors when enhancing or deploying malformed classes. These should be easily fixable by modifying your object-id classes to conform to the JDO specification rules.
 - Changed `OnDemandForwardResultList` to not use weak or soft references, but instead to optionally use a scrolling window to prevent memory growth as large result sets are iterated.

2.5.2 -- 4 July 2003

- Bugfixes
 - The repackaged `concurrent.util` APIs have been included in the released jars.
 - Fixed potential rounding bug where the fractional parts of a `Date` field can be doubled when using JDK 1.4.1.
 - Fixed problem where `AutoIncrementSequenceFactory` was not working for SQL Server.
- Notable changes
 - Changed the `ConnectionRetainMode` fix that was made in 2.5.1 to not actually close the `PersistenceManager`, replicating the behavior of 2.5.0 and earlier. This means that session beans that return live JDO objects without detaching them or copying them into data transfer objects will continue to function as with 2.5.0 and earlier releases. It is likely that Kodo 3.0 will deal with this differently, possibly including a mode to allow the current, more lenient behavior.

2.5.1 -- 4 July 2003

- New Features
 - Added ability to use database-specific outer join syntax. Coded Oracle 8i outer joins into Oracle dictionary.
 - Borland Enterprise Server is now supported, meaning that the `AutomaticManagedRuntime` class knows about where Borland puts its transaction manager in JNDI. In addition, the J2EE tutorial has been updated with detailed deployment instructions for Kodo as a JCA Resource Adapter.

- Bugfixes
 - Eclipse/WSAD plugin ClassLoader problems resolved. Please update the plugins/com.solar.../kodo-jdo.jar to the latest release.
 - Fixed ConnectionRetainMode=persistence-manager to correctly close resources when used in a container-managed transaction context. This fix may cause applications that use session beans but do not properly (serialize | clone | makeTransient) persistence-capable objects returned from the session beans to throw exceptions stating that a PersistenceManager has been closed. This can only be an issue if your session bean is deployed to the same JVM as the EJB client code.
 - Deadlocking problem with upgrading read locks in data cache has been resolved.
 - Optimistic lock version problem when RetainValues is false was resolved.
 - Connection leak problem in AutoIncrementSequenceFactory was resolved.
 - Improved performance of JDO class initialization in environments with potentially slow classloaders, such as JBoss.
- Notable changes
 - The included distribution of Apache Commons Logging is now 1.0.3. Be sure to update your classpath accordingly as there were some difficult to diagnose configuration bugs in 1.0.2. The JCA rar file has been updated with the newer version.
 - The UsePreparedStatements option has been removed: prepared statements are now always used for all drivers.
 - Made all queries using unbound variables use SELECT DISTINCT.
 - Kodo once again forces the prepared statement pool size to zero when using Microsoft's JDBC driver. You can prevent Kodo from doing this by setting the com.microsoft.jdbc.sqlserver.SQLServerDriver.nopool system property. See http://bugzilla.solarmetric.com/show_bug.cgi?id=501 for details.

2.5.0 -- 5 June 2003

- New features
 - Custom fetch groups are now supported. See the fetch group documentation and the FetchGroups configuration documentation for more information.
 - Participation in a global XA-compliant transaction is now possible in a managed environment.
 - Multi-table mappings now permit different tables to have different primary key column names.
 - The ProxyManager now includes capabilities to proxy user-defined mutable field types that are not part of the JDO specification.
 - Kodo now supports auto-increment columns when using datastore identity. See the sequence-factory-class metadata extension and SequenceFactoryClass configuration property documentation for usage details.
 - New flush API allows the modifications made in a transaction to be incrementally flushed to the database before transaction commit time. See the **com.solarmetric.kodo.runtime.KodoTransaction.flush()** JavaDoc for details.
 - Added subclasses of JDOUserException for special cases that are of interest: **com.solarmetric.kodo.runtime.OptimisticLockException** and **com.solarmetric.kodo.runtime.ObjectNotFoundException**.
 - New Kodo J2EE integration tutorial. See the J2EE documentation as well as the source code before proceeding. Cur-

rently, the tutorial includes instructions for WebLogic 6.2 and higher, SunONE Application Server 7, WebSphere 5, and JBoss 3.x.

- The association between a PersistenceManager and a JDBC Connection can now be configured. The default behavior is the same as in earlier versions of Kodo -- connections are obtained on-demand. Additionally, Kodo can be configured to retain a connection for the duration of a transaction (both optimistic and pessimistic) or for the duration of a PersistenceManager's life cycle. This behavior is controlled with the `com.solarmetric.kodo.impl.jdbc.ConnectionRetainMode` configuration property.
- Enhancement-time validation of JDO metadata has been improved. This may result in errors next time you recompile and re-enhance your persistence-capable classes.
- Modified persistence-capable classes can be grouped by class before being flushed to the data store, increasing the potential for performance benefits due to statement batching. See the `ClassGroupStateManagerSet` documentation for details about this option.
- Added direct support for custom collections and maps that implement `ProxyCollection` or `ProxyMap`, and for fields that implement `Proxy`.
- Informix IDS is now a supported database.
- Second-class objects that are externalizable to Strings can now be stored to string fields. See the `Storing Second Class Objects via Stringification` documentation for more information.
- Data caching framework now caches JDOQL queries. See the `Kodo JDO Query Caching` section for more details.
- Data caching framework includes semantics for specifying a timeout for a given class. See the `Metadata` documentation for more details.
- Different classes can use different `PersistenceManagerFactory` caches, allowing for varying cache policies on a per-class basis.
- A transaction event listener framework has been created. This framework allows listeners to be notified of transaction begin, commit, and rollback events on a per-`PersistenceManager` level, and of transaction commits on a per-`PersistenceManagerFactory` level. Additionally, this framework allows transaction commit notification to be propagated to remote `PersistenceManagerFactory` objects. See **event notification framework** documentation for more information.
- The schema manipulation done by the `SequenceFactory` to initialize any database-specific tables to store sequence information is now done when the `schematool` is run, rather than at runtime.
- Queries have received a major overhaul. Queries now support unbound variables, Collections as parameters to generate SQL IN (...) clauses, traversing fields of persistence-capable parameters, and more. The SQL produced by queries is also much more efficient.
- The `Query FilterListener` API has changed, and the default set of available `FilterListeners` has been enhanced with a few new and powerful extensions. Some of the old extensions have been deprecated, so check the `Query Extensions` section of the documentation for details on the new extensions framework. Additionally, it is unlikely that existing custom query extensions will continue to work.
- Added the `com.solarmetric.kodo.impl.jdbc.UseSQL92Joins` configuration property. Set this property to `true` to use SQL 92-style joins in queries, including left outer joins where appropriate. (This is the default value.) You can also set this property on an individual query instance; see the `com.solarmetric.kodo.impl.jdbc.query.JDBCQuery` class Javadoc for details.
- `PersistenceManager.newQuery(Class)` and `PersistenceManager.getExtent(Class)` can now take an interface as a parameter, even when multiple separate inheritance hierarchies implement the interface and exist in the data store. Ordering for queries will work as expected, but it should be noted that an ordered query that is executed against multiple tables will result in partial loss of large result set support, such that attempting to access element N in the Collection returned from `Query.execute()` will force the results 0..N-1 to be instantiated so that an in-memory comparison of the homonegous ob-

jects can take place.

- The new properties `com.solarmetric.kodo.ResultListClass` and `com.solarmetric.kodo.ResultListProperties` can now be used to specify a custom implementation of the `CustomResultList` interface that will be used to hold queries.

Notable changes

- The distributed data cache framework has been changed to use the transaction event listener framework to communicate commit information to remote JVMs. This means that deployments that use distributed caching must set up the `com.solarmetric.kodo.RemoteCommitProviderClass` and `com.solarmetric.kodo.RemoteCommitProviderProperties` configuration properties appropriately. Additionally, communication-related configuration properties in the `com.solarmetric.kodo.DataCacheProperties` must be removed.

For example, to configure Kodo to use JMS for distributed commit notification, your properties would look like so:

```
com.solarmetric.kodo.DataCacheClass: com.solarmetric.kodo.runtime.datacache.plugins.CacheImpl
com.solarmetric.kodo.RemoteCommitProviderClass: com.solarmetric.kodo.runtime.event.impl.JMSRemoteCommitProvider
com.solarmetric.kodo.RemoteCommitProviderProperties: Topic=topic/KodoCacheTopic
```

To configure Kodo to just share commit notifications among `PersistenceManagerFactories` in the same JVM, your properties would look like so:

```
com.solarmetric.kodo.DataCacheClass: com.solarmetric.kodo.runtime.datacache.plugins.CacheImpl
com.solarmetric.kodo.RemoteCommitProviderClass: com.solarmetric.kodo.runtime.event.impl.SingleJVMRemoteCommitProvider
com.solarmetric.kodo.RemoteCommitProviderProperties: Topic=topic/KodoCacheTopic
```

- The `UDPCache` distributed data cache plug-in has been removed. People interested in using UDP for cache invalidation should implement the `com.solarmetric.kodo.runtime.event.RemoteCommitProvider` interface.
- The `DataCache` interface and associated implementations have been overhauled in a number of ways. As a result, it is unlikely that previously-created `DataCache` implementations will work with Kodo 2.5.
- When `IgnoreCache` is set to `false` and a query is executed after modifications to instances that are in the query's access path, Kodo may automatically flush all modifications in the current transaction to the database, and performs the query against the data store. The behavior depends on numerous settings; see `FlushBeforeQueries` for details. Previous releases of Kodo evaluated these types of queries in-memory, which can incur a considerable performance penalty.
- Added a validation to ensure that ordering strings explicitly use either `ascending` or `descending` correctly, and do not specify any other values for the ordering.
- Eclipse/WSAD plugin has changed to 1.0.1. The Kodo view is now located in the Java grouping, as opposed to Debug. The plugin is now compatible with Eclipse 2.1. In addition, the required jars in the `plugin.xml` has been changed to include Jakarta's `lang` jar (included with the distribution). The plugin should be reinstalled (remove the old `com.solarmetric...` directory and reinstall according to the documentation).
- NetBeans/SunONE plugin users should install the Jakarta `lang` jar from the distribution into the `lib/ext` directory of their installation. In addition, `serp.jar` should be removed as it is now part of the regular distribution and is no longer needed. See the full list of required jars in the SunONE/NetBeans portion of the documentation.
- Bugfixes
 - Fixed `datacache` issue with over-eager loading of relations.

- Fix for potential inefficiency when many threads concurrently access the data cache.
- Fixed finalization bug in connection pooling that allowed closed connections to be returned from the connection pool.
- Placed subselects in generated SQL for isEmpty on right side of expression to placate DB2.
- Using persistence-capable parameters that implement Collection or Map in a JDOQL query now works.
- Assorted minor bugfixes and error message improvements.
- Method name misspelling in `com.solarmetric.kodo.impl.jdbc.SQLExecutionListener` has been fixed. As a result, implementations of this interface must be changed to use the correctly-spelled method name.
- Merged 2.4.3 bugfixes. See below.
- Fixed many query bugs.

2.4.3 -- 26 March 2003

- Notable changes

Bugfixes

- Included a new version of serp that resolves issues with weak and soft references that can lead to memory leaks. Be sure to copy the new serp jar into your lib dir as well as the new Kodo jars.
- Fixed a bug in `ClassDBSequenceFactory` to address potential concurrency issues that could lead to a deadlock while obtaining new ID values.
- Fixed `AbstractDictionary` to deal with null `Locale` objects properly.

2.4.2 -- 26 Feb 2003

- Notable changes

- The SunONE Studio / NetBeans plugin module has moved into release status. Existing module users should un-install and re-install the module.
- The Eclipse / WSAD plugin has moved into release status. *Note that the plugin folder structure has changed to reflect this change.* Existing plugin users should remove the old folder, install the new folder, and update the `plugin.xml` accordingly. Included in this new version are changes in classpath and project resolution.

Bugfixes

- Mutating a `Date` field via deprecated `setDate()`, `setHour()`, etc. now properly dirties the owning object.
- Fixed `LocalCache` synchronization issue.

2.4.1 -- 26 January 2003

- New features

- New subclass provider implementation option simplifies using an integer lookup value to store subclass information in the database. Additionally, the source for this implementation is included in the release, so creating a custom subclass provider is simpler.
- We now set the default transaction isolation level to `TRANSACTION_SERIALIZABLE` when using DB2. This is necessary in order for datastore (pessimistic) transactions to lock rows correctly.
- Added a new SequenceFactory: `com.solarmetric.kodo.impl.jdbc.schema.ClassDBSequenceFactory` which provides class sensitive table-based id sequences. To use the new sequence factory, existing sequence tables need to be dropped to be mapped to the differing table structure.
- Added a table name option to `com.solarmetric.kodo.impl.jdbc.schema.DBSequenceFactory` to map sequences to. To set this option, add the option `tableName=yourname` to `com.solarmetric.kodo.impl.jdbc.SequenceFactoryProperties` property when configuring your `PersistenceManagerFactory`.
- Persistent types can once again be enumerated by using the `com.solarmetric.kodo.PersistentTypes` property. This property is optional, but help to avoid classloader issues when deploying to an application server.

Bugfixes

- Fixed data caching plug-in to not enlist objects with `can-cache=false` when loading existing data from the database.
- Fixed bug in metadata parsing algorithm that could cause classloader problems in application servers
- Fixed `SQLServerDictionary` to work around `SQLServer`'s issues with setting null BLOBs via `PreparedStatement.setNull()`.
- Datastore locking (i.e., pessimistic locking) is now supported for Sybase. Note that the connection property `"BE_AS_JDBC_COMPLIANT_AS_POSSIBLE=true"` must be set, either in the `ConnectionURL` or the `ConnectionProperties` properties. See the `SybaseDictionary.java` source file for more details. This requires the Sybase JDBC driver version 4.5 or higher.

Notable changes

- Removed the `AutoReturnTimeout` property. The Kodo pooling `DataSource` no longer reclaims expired connections.

2.4.0 -- 13 Dec 2002

- New features
 - Pre-release versions of plugins for SunONE Studio, NetBeans, Eclipse, and WebSphere Studio are now available. See the documentation on installation and usage instructions.
 - Kodo JDO Enterprise Edition and the Kodo JDO Performance Pack are now bundled with a cache plug-in that supports Tangosol Coherence cache products. See the datastore cache documentation for details.
 - Added `evictAll(Class)` and `evictAll(Extent)` method calls to `PersistenceManagerImpl`. These methods are useful for clearing often-updated objects in pooled `PersistenceManager` configurations.
 - Added the capability of loading `ResultSet` objects (or any other stream of data) into `PersistenceCapable` objects associated with a `PersistenceManager` via application-defined logic.
 - Added metadata extensions for specifying custom `ClassMapping` and `FieldMapping` values for particular classes and fields.
 - Added class-level metadata extension to exclude certain classes from the `PersistenceManagerFactory` cache.

- Added property for configuring the how long to wait before testing connections that have been put into the pool.
- Simplified the process of defining custom subclass indicator behavior.
- When supported by the underlying JDBC driver, Kodo will now use PreparedStatements and batch updates whenever possible for very significant performance benefits. See the documentation for the `com.solarmetric.kodo.impl.jdbc.UsePreparedStatements` and `com.solarmetric.kodo.impl.jdbc.UseBatchedStatements` properties.
- Inverse one-to-one mappings are now supported. The field can now reside on either table corresponding to the related objects. If both sides of an one-to-one are marked as having an inverse, one field should be designated as read-only to indicate to the system the owning class and table for the given relational field.
- Logging is now done through the Apache commons project, which offers the ability to use an underlying logging mechanism, such as Apache Log4J, JDK 1.4's native logging, or simple file/stdout logging. It is now necessary to include the new `jakarta-commons-logging-1.0.2.jar` in the CLASSPATH. See the **Logging** chapter.
- Added a pluggable `SQLExecutionManager` architecture, which allows the developer to override the mechanism by which SQL is issued to the database. See the documentation for the `com.solarmetric.kodo.impl.jdbc.SQLExecutionManagerClass` property.
- There is now an option to automatically refresh the database schema during runtime, allowing the developer to skip the `schematool` step. See the documentation for the `com.solarmetric.kodo.impl.jdbc.SynchronizeSchema` property.
- Properties may be specified for a Driver with the `com.solarmetric.kodo.ConnectionProperties` property.
- A `javax.sql.DataSource` may be specified in the **`kodo.ConnectionDriverName`** property, which will be customizable with bean-like entries in the `com.solarmetric.kodo.ConnectionProperties` property.
- The default transaction isolation for a JDBC connection can be overridden with the **`com.solarmetric.kodo.impl.jdbc.TransactionIsolation`** property.
- Kodo JDO now distributes a single jar for both the enterprise and standard edition, as well as datacache and query extensions.
- The `rd-metadatool` can now be used to generate default JDO metadata for classes.
- The **`rd-schemagen`** tool used for reverse mapping classes from a schema has now been tested with the following databases: Hypersonic SQL 7.1, SQLServer (MS Beta 2 JDBC driver), Sybase, Oracle (9.0.1 JDBC driver), DB2, Postgres (7.3 Beta 3 JDBC driver).

Bugfixes

- `default-fetch-group="false"` is now respected for fields that default to `default-fetch-group="true"`.
- Traversing orphaned relations in data cache now behaves in the same way as traversing orphaned db relations -- the invalid relation is set to null.
- Changing a field to the same value as it was originally set to no longer constitutes dirtying that field. This means that subsequent flushes to the database will not necessarily re-write the same data.
- Class loading is performed in accordance with section 12.5 of the JDO specification.

Notable changes

- The API for implementing a `SequenceFactory` has changed. See the API documentation for **`com.solarmetric.kodo.impl.jdbc.SequenceFactory`**.
- Persistent types are no longer enumerated, either in the data store or in a property. Classes are now dynamically added

upon class initialization, via the JDOImplHelper class registration process. The `-register` and `-unregister` options to `schematool` are no longer needed. The `JDO_SCHEMA_METADATA` table is no longer used and can be dropped.

2.3.4

- New features
 - The R&D schema generator can now accept a list of tables to generate.
 - The R&D reverse mapping tool has additional options for using foreign key names to generate relation names, generating primitive wrapper-type fields if a column is nullable, and allowing primary keys on many-to-many join tables.
- Bugfixes
 - Fixed problems with many-to-many relations between tables that use vertical inheritance.
 - Fixed bug in `schematool` that caused it to not generate primary key columns in subclass tables when using `datastore identity + custom names + vertical inheritance`.
 - Fixed `serp` library conflict between reverse mapping tool and main Kodo libraries.
 - Fixed a reverse mapping tool bug in which column names that conflicted with Java keywords would result in the generation of uncompileable Java classes.
 - Fixed problem that caused `read-only` flag to be ignored in many-to-many relations.
 - Multi-table inheritance deletes are now performed from the leaf table in the inheritance chain up to the base table. Inserts are performed from the base table down to the leaf. This supports the common referential integrity model of establishing a foreign key relation from inherited tables to their parent tables.

2.3.3

- New features
 - The R&D schema generator can now accept a list of tables to generate.
- Bugfixes
 - Fixed `null-value="default"` behavior.
 - Fixed bugs that prevented removal of map elements through the key set, entry set, and values collection.
 - Added more validation on `static/final` fields to metadata.
 - Fixed memory leak in `serp` regarding soft and weak collections backed by `HashSets`.
 - Multi-variable query issues resolved.
 - Fixed bug that could cause optimistic lock version numbers to be incremented before successful transaction commit.
 - The R&D reverse mapping tool now handles Oracle `DATE` columns correctly.

2.3.2

- New features
 - The new `com.solarmetric.kodo.CacheReferenceSize` property dictates a number of hard references to cached objects that the `PersistenceManager` will retain, in addition to its soft cache.
 - Added 'all' option to unregister action of schematool Ant task. This option allows all classes in the current persistent types list to be unregistered, regardless of whether or not those classes are currently in the classpath.
 - Added `com.solarmetric.kodo.UseSoftTransactionCache` to configure whether or not Kodo should maintain soft references to transactional items that have not been dirtied. This now defaults to false; previous versions of Kodo always used soft references for non-dirty transactional items.
- Bugfixes
 - The `jdodoclet` task no longer creates JDO metadata entries for final or static fields, or for transient fields that do not have a `jdo:persistence-modifier` tag.
 - The `jdoc` task no longer attempts to enhance classes that have already been enhanced.
 - Several default property values were being set improperly.

2.3.1

- New features
 - Smart proxies for set and map fields. Smart proxies better optimize database updates when persistent set and map fields are modified.
 - TCP, JMS-based distributed `DataCache` implementations.
 - All `DataCache` implementations now use an LRU cache with a configurable maximum size.
 - Customizable tracked instance proxies.
 - Alpha release of upcoming reverse mapping tool for creating persistent class definitions, metadata, and mapping extensions from an existing schema.
 - Cache object `com.solarmetric.kodo.runtime.datacache.PMFactoryCache` is now named `com.solarmetric.kodo.runtime.datacache.plugins.LocalCache`.
- Bugfixes
 - A Query with an unspecified filter defaults to a filter of "true", rather than "false".
 - A Query with an unspecified candidate Extent but a specified candidate Class will automatically create an Extent of the appropriate type with subclasses turned on.
 - Various bugs related to compiled queries with null arguments or no parameters have been resolved.
 - Various `InstanceCallbacks` interface bugs have been resolved.
 - Ant schematool and jdoc tasks deal with the Ant `ClassLoader` system better.

- Notable changes
 - Made `GenericDictionary.toSQL()` and `GenericDictionary.fromSQL()` methods `final`. Subclasses of `GenericDictionary` that must change the behavior of SQL generation or parsing should do so by overriding the appropriate `xxxToSQL()` or `xxxFromSQL()` methods instead. Note that the source for all our dictionary classes is now available in the Kodo JDO distribution.

2.3.0

- New features
 - JDO specification version 1.0 support.
 - Highly flexible multiple-table inheritance model now supported. See the multi-table class mapping documentation for details.
 - Support for large result sets when using any JDBC 2.0+ driver that supports ResultSets of type `TYPE_SCROLL_INSENSITIVE`. Return values from all `Query.executeXXX()` methods will be an instance of `java.util.List`, which can then be used for efficient random access.
 - DataCache API batches distributed updates, facilitating custom processing of distributed cache invalidation.
 - DataCache implementation loads data read from the data store into the cache as well as data being written to the data store.
 - JDBC back-end customizability is improved, allowing for custom field and class mappings and much finer-grained control of generated SQL.
 - Kodo JDO now supports extending JDOQL with custom tags. A number of default extensions, including substring searches and case-insensitive searches, are included by default with Kodo JDO. For more on this feature, see the query extensions documentation.
 - Support for IBM WebSphere, and other application servers that do not provide a `TransactionManager` through a JNDI lookup.
 - Source code release for various utility classes under the `source/` directory.
 - Deprecated the `srcDir` attribute of `jdoc` and `schematool`: the nested fileset no longer needs to be relative to an absolute directory.
 - Added a new method to `ExtentImpl` that returns a list containing all objects described by the extent.
- Bugfixes
 - Resolved a problem with large (> 5000 bytes) BLOBs being stored in Oracle.
 - Fixed problem with compiled queries and null parameters returning empty result sets. Currently, when null parameters are used, prepared statements are bypassed and custom SQL is generated instead. In a future release, a new prepared statement that uses `IS NULL` will be generated on-the-fly.

2.2.5 May 6, 2002

- New features

- The String serialization of ObjectIds.Id now uses a '-' as the delimiter, as the previous choice of the '#' made it difficult to use the serialization in a JSP without re-encoding it.
- Released ant tasks for JDO enhancement, the SchemaTool, and an XDoclet task for generating .jdo metadata files from java source code comments.
- Integration features for the upcoming JBuilder 7.
- Bugfixes
 - Fixed issue with managed transaction rollbacks. See bug #157 for details.
 - Fixed problem with prepared statements and in-memory queries. See bug #161 for details.

2.2.4 SP1 April 19, 2002

- Bugfixes
 - Resolved issues when using null parameters and compiled queries.

2.2.4 April 17, 2002

- New features
 - Support for java.math.BigInteger and java.math.BigDecimal
 - Added support for using packagename.jdo as the package's JDO metadata file, where "packagename" is the last section of the resource's package. E.g., a java class named com.solarmetric.mypackage.MyClass can now use a metadata file named mypackage.jdo.
 - Query.compile() will now create and use a PreparedStatement.
 - Kodo JDO now supports both single-JVM and distributed caching of persistent data.
 - It is now possible to extend the PersistenceManagerImpl and the EEPersistenceManager.
 - Added support for serialization/deserialization of an object id to/from a String.
 - Added an example of using Kodo within JSPs in the samples/jdo/jsp/ directory.
- Bugfixes
 - Resolved inefficient behavior when executing a query that returns objects that have already been loaded but are hollow (bug 116).
 - Fixed NotSerializableException when trying to bind an instance of EEPersistenceManagerFactory into JNDI (bug 117).
 - Fixed problem where changing any of the configuration values in a PersistenceManagerFactory changed those values for all PersistenceManagerFactory instances on the system (bug 131).

2.2.3 March 4, 2002

- New features
 - New and improved documentation is now available at docs/manual.html. Enjoy!
 - Added code to check parameter count against declared parameter count when executing queries.
 - Partial support for the Java Connector Architecture is now available in the Enterprise Edition. This permits simple configuration of Kodo JDO using an application server's JCA configuration tools.
 - PersistenceManagerFactory instances can now be created from a Properties.
 - Guaranteed that SQL statements corresponding to object modifications (insert, update, delete) occur in the order that the modifications were performed in the PersistenceManager. If an object is modified multiple times, it will remain in the position that it was in after its first modification. When committing, we now traverse this list in order, so it is possible to do things like delete an object and then add a new object with the same id in a single transaction.
 - Added optional '-outfile filename' option to schematool. If specified, the SQL statements necessary to perform the schema modification will be appended to filename. No changes will be made to the database itself. This is useful when database modification is not permitted, or for post-processing the SQL generated by Kodo JDO with an external tool.
 - Changed schematool to print a warning when an array, collection, or map field is implicitly made persistent because of the rules of the spec. This often leads to undesirable behavior, as the default mode of insertion is to serialize the array/collection/map into a BLOB field, which is more often than not the desired behavior.
 - Both 'kodo' and 'tt' are now supported vendor tags. No collision checking is performed, so you should probably use just one.
 - Added support for Hypersonic free file-based JDBC driver
 - Added a new database preference: db/schema-name. If set, this value will be used in calls to DatabaseMetaData.getTables().
- Bugfixes
 - Made queries take into account changes in the current transaction if IgnoreCache == false.
 - Made extents pay attention to changes made in the current transaction
 - Changed methods that are part of javax.jdo interfaces to never throw anything but JDOExceptions. See bug 69.
 - Resolved problem with listing table names when multiple database users should each have their own set of tables. See bug 77.
 - Only invalidate the connection and not return it to the pool if the Connection name contains "postgres". See PostgreSQL bugfix in 2.2.2 section for more details.
 - Fixed a problem where executing 'jdoc' on a package.jdo that contains both app id and datastore id classes causes a failure.
 - Improved error messages.

2.2.2 February 14, 2002

- New features
 - Added a duplicate column check to SQL INSERT and UPDATE query generation methods. If a duplicate column name is encountered and the values are also duplicates, then life proceeds happily along. If duplicates are found and the values differ, a JDOUserException is thrown. This permits using schema mappings in which a column is used both as a primary key

and a foreign key.

- Bugfixes
 - Resolved potential deadlocks. See bug 42.
 - Added mechanism for controlling date precision when constructing SQL statements. See bug 6.
 - Fixed schematool strangeness when using table name metadata extensions. See bug 54.
 - improved error-reporting in exceptions thrown when invalid data is added to proxy collections/maps.
 - Fixed bug in which persistent-deleted objects were not containing the correct values on rollback if RetainValues was set. This fix makes persistent-deleted objects transition to hollow instead of performing any rollback.
 - Transaction.commit() and Transaction.rollback() now throw a JDOUserException instead of an IllegalStateException when a transaction is not active. See bug 44.
 - Because of a probable Postgres JDBC driver bug, changed connection pooling to not recycle connections that have been involved in a transaction.
 - Resolved a VerifyError that occurred when a non-primitive, non-String object was used as part of an object's primary key.
 - Resolved a situation in which the number of connections needed to load a single object from the data store was proportional to the depth of persistence-capable fields in the tree of default fetch groups. That is, if A has a relation to B called b, and B has a relation to C called c, and b is in A's default fetch group, and c is in B's default fetch group, then three connections were needed in order to load an A.
 - Fixed bug in which queries on date fields occasionally threw exceptions.
 - Fixed obscure bug in makeDirty(). If using data store transactions and setting a JDO field without first having loaded the field (either implicitly by having the field in the default fetch group, or explicitly), then the field would not be set when InstanceCallbacks.jdoPostLoad() was invoked. Additionally, nontransactionalRead must have been set to true for this problem to occur.
 - Fixed a bug that caused queries to fail in certain Tomcat configurations. See bug 35.
 - Added writeReplace() methods to fix issues with serialization of dates and collection types retrieved from data stores.
 - Fixed bug in which jdoNewInstance(StateManager,Object) method was only being added to base application identity classes.
 - com/solarmetric/kodo/runtime/PersistenceManagerImpl.java: improved error reporting when validating and making persistent objects that are not managed by the current PM.
- Notable changes
 - Made default table type for MySQL be Berkley DB, which has real transactional capabilities
 - Set the default to warn on persistent type failures, rather than throw an exception.

2.2.1 November 1 2001

- New features
 - IBM DB2 UDB 7.2 is now supported.

- All datastore identity classes now use the `com.solarmetric.kodo.util.ObjectIds.Id` object ID type rather than individual dynamically loaded classes.
- Performance enhancements.
- Bugfixes
 - Old versions of MySQL for Windows are now supported.
 - A new algorithm for auto-generation of table/column/index names that is much less likely to generate naming collisions is now available.
 - Fixed `PersistenceManager.refreshAll()` behavior when no transaction is active.
 - New persistent objects, first class children, etc. are correctly dealt with when created in `jdoPreStore()`.
 - Queries that perform multiple `contains()`, `containsKey()`, or `containsValue()` clauses &&'d together for different values on the same collection/map now work.
 - The PM will throw some subclass of `JDOFatalException` on commit if and only if the transaction is also automatically rolled back.
- Notable changes
 - One-to-one mappings are dealt with more efficiently, reducing the number of database accesses and therefore improving performance.
 - Changed resource-loading and class-loading to use the current thread's context's class loader, rather than the system class loader. This makes deployment to a web application container much easier.
 - A single class is now used as the ID class for all persistent types managed with data store identity.

2.2.0 October 5, 2001

- New features
 - Application identity is now supported.
 - Preview release of tool to generate a class suitable for use as an application-identity object id class, complete with an appropriate `equals()` method, a corresponding `hashCode()` method, and a `toString()` method. For more information and usage, run `'java com.solarmetric.kodo.tools.appid.ApplicationIDTool'`.
 - Improvements have been made to common error messages, and inappropriate exception types have been replaced with more useful ones.
 - New library: `kodo-jdo-runtime.jar`. This library contains all the class files necessary for run-time use of Kodo JDO.
 - Enhanced mapping customizations for mapping application-identity pk fields (see `docs/existing-schema.html`)
 - Various minor performance enhancements
- Bugfixes
 - `PersistenceManager` refresh methods behave correctly when invoked from outside the context of a transaction. Note that the noargs `refreshAll ()` call behaves as designated by the JDO javadoc, not as designated by the 0.95 specification.

- SQL generation for statements that insert decimals (floats and doubles) now always use United States notation (3.14159 for example).
- Assorted minor bugfixes.
- Notable changes
 - Major redesign of the refresh mechanism.

beta 2.1 July 15, 2001

- New features
 - The null-value attribute on field metadata is supported.
 - BLOB mappings are supported; any serializable field can now be persisted. (Note: PostgreSQL does not support BLOB mappings)
- Bugfixes
 - `jdoPreStore()` is no longer called on deleted instances.
 - Fixed a `NullPointerException` that could occur when softly-cached instances were garbage collected by the JVM.
 - Indexes were not being created on fields marked with the 'column-index' metadata extension.
 - Fixed a bug that prevented retrieving CLOB values with Oracle.
 - Fixed a bug that caused a `SQLException` with fields set to empty strings or chars with a 0 value on PostgreSQL.
- Notable changes
 - The `SQLTypeMap`, used in `DBDictionaries`, changed slightly.
 - The Kodo User Guide chapter on Metadata has been updated to include information on the new 'blob' metadata extension for explicitly marking fields that should be stored in serialized form.

beta 2 July 10, 2001

- New features
 - Maps with user-defined persistent object values can be persisted (n-many relations).
 - Static inner classes can be persisted.
 - Queries support the use of `containsKey()` and `containsValue()` for Map fields.
 - Queries support ordering declarations.
 - The `SchemaTool`'s schema migration capabilities have been enhanced.
 - The `SchemaTool` offers the option of automatically maintaining the list of persistent types for the system.
 - Schema generation can be customized through JDO metadata.

- The standard `javax.sql.DataSource` is used to obtain connections.
- Connection pooling has been enhanced, and new pooling parameters have been added (timeout time, autoreturn time).
- The `ObjectId` helper class has been introduced to map opaque JDO OID values to and from primitive long values.
- `PersistenceManagerFactories` can be stored in JNDI, including JNDI trees that are replicated over multiple JVMs.
- `PersistenceManagers` can transparently synchronize with global J2EE Transactions (Kodo Enterprise Edition beta only).
- Bugfixes
 - Row-level locking is now performed within pessimistic Transactions.
 - Object ID generation is now done using the database by default.
 - Globally unique primary key values are no longer required (per-class only).
 - Inserting new instances of classes mapped to an existing schema without a class indicator column no longer throws a `NullPointerException`.
 - Numerous minor fixes.
- Notable changes
 - Users of previous beta versions of Kodo should scan the user guide in the `docs/` directory for new information included with this release.
 - The `schematool` can now automatically maintain a list of persistent classes; the `persistent-types` array in `system.prefs` is not needed. This is covered in the Database Setup chapter of the user guide.
 - The syntax for using the `schematool` has changed. This is covered in the Database Setup chapter of the user guide.
 - The syntax for mapping classes to existing database tables has changed. This is covered in the Using Existing Schema chapter of the user guide.
 - The Runtime Use chapter of the user guide covers new runtime options available, such as JNDI storage of the `JDBCPersistenceManagerFactory` and safe conversion of JDO OID values to and from primitive long values.

Index

A

- ABS function, 87
- ACID, 79, 226
 - (see also transactions)
- AggregateListeners, 397
- aggregates, 235, 244
 - (see also JDOQL)
 - avg, 244
 - return type, 246
 - count, 244
 - return type, 246
 - grouping, 245
 - having, 245
 - SupportsHaving, 432
 - max, 244
 - return type, 246
 - min, 244
 - return type, 246
 - sum, 244
 - return type, 246
- aliases, 247
 - (see also projections)
- allocating sequence values, 319
 - (see also Sequence)
- annotations, 28
 - Basic, 35
 - Embeddable, 30
 - Embedded, 36
 - EmbeddedId, 35
 - Entity, 29
 - EntityListeners, 31
 - Enumerated, 143
 - GeneratedValue, 34
 - Id, 34
 - IdClass, 30
 - Lob, 143
 - ManyToMany, 40
 - ManyToOne, 36
 - MapKey, 41
 - MappedSuperclass, 30
 - OneToMany, 38
 - OneToOne, 39
 - OrderBy, 41
 - Transient, 34
 - Version, 35
- Ant, 607
 - application identity tool task, 609
 - configuration options, 607
 - enhancer task, 609
 - mapping tool task, 610
 - reverse mapping tool task, 610
 - schema tool task, 611
- applets, 582
- ApplicationIdentity, 212

- (see also application identity)
 - (see also supported options set)
- application identity, 187, 187, 195, 221
 - (see also identity)
 - checking for change support, 212
- application identity tool, 456
 - Ant task, 609
- array
 - metadata, 197
 - (see also persistent fields)
- Array, 212
 - (see also persistent fields)
 - (see also supported options set)
- ArrayList, 212
 - (see also persistent fields)
 - (see also supported options set)
- arrays
 - as persistent fields, 182
- ascending, 241
 - (see also Query, ordering)
- association table, 298
- atomicity, 79, 226
 - (see also transactions)
- attachCopy, 219
 - (see also detachment)
- attachment (see detachment)
 - behavior, 577
- AutoClear, 397
- AutoDetach, 398
- avg, 244
 - (see also aggregates)

B

- backwards compatibility, 398
- Basic, 35
- Batching
 - TestBatchSupport, 436
- BETWEEN expressions, 84
- bidirectional relation, 301, 303
 - InverseManager, 405
- bidirectional relations, 39, 296
 - (see also persistent fields)
 - automatic management, 458
 - mapping, 158
- BinaryCompatibility, 213
 - (see also binary compatible)
 - (see also supported options set)
- binary compatible
 - checking for implementation support, 213
- BLOB
 - BlobTypeName, 433
 - field mapping, 291
 - MaxEmbeddedBlobSize, 435
 - UseGetBytesForBlobs, 432
 - UseGetObjectForBlobs, 432
 - UseSetBytesForBlobs, 433
- Borland, 532
- bound variables (see variables)
- Broker

BrokerImpl, 398
 BrokerFactory, 398
 BrokerImpl, 398
 ByteIdentity, 191
 (see also single field identity)

C

catching

cron-style invalidation, 562
 DataCache, 403
 data cache, 560
 extension, 574
 data cache MBean, 600
 DataCacheTimeout, 403
 DataStoreCache, 257
 DynamicDataStructs, 404
 issues and limitations, 575
 log messages, 417
 named caches, 562
 QueryCache, 409
 query cache, 566
 extension, 574
 QueryCompilationCache, 409
 query compilation cache, 576
 RemoteCommitProvider, 410
 size, 561
 tangosol integration, 572
 timeout, 562
 candidate class, 230, 233
 (see also Extent)
 (see also Query)
 candidate objects, 233
 (see also Query)
 CascadeType, 37
 ChangeApplicationIdentity, 212
 (see also application identity)
 (see also supported options set)
 CharIdentity, 192
 (see also single field identity)
 checkConsistency, 223
 (see also PersistenceManager)
 class
 persistent (see persistent classes)
 class loading (see ClassResolver)
 CLASSPATH, 330, 347
 ClassResolver, 398
 clear, 77
 (see also EntityManager)
 ClearCallback (see InstanceCallbacks)
 ClearLifecycleListener, 186
 (see also lifecycle callbacks)
 CLOB
 ClobTypeName, 433
 field mapping, 290
 MaxEmbeddedClobSize, 435
 UseGetStringForClobs, 433
 UseSetStringForClobs, 433
 closePersistenceManagerFactory, 213
 (see also JDOPermission)

CMT, 228
 code formatting, 395
 collection
 metadata, 198
 (see also persistent fields)
 Column
 in mapping metadata, 123
 (see also mapping metadata)
 column
 in mapping metadata, 268
 (see also mapping metadata)
 Columns, 510
 (see also mapping metadata)
 Compatibility, 398
 CONCAT function, 87
 configuration, 393
 command line, 394
 log messages, 417
 of JDBC properties, 412
 of JDO properties, 397
 of Kodo properties, 397
 plugins, 395
 runtime, 393
 Connection2DriverName, 399
 Connection2Password, 401
 Connection2Properties, 401
 Connection2URL, 402
 Connection2UserName, 402
 ConnectionDecorators, 412
 ConnectionDriverName, 208, 399, 423, 426
 (see also connections)
 (see also PersistenceManagerFactory)
 ConnectionFactory, 209, 399, 426
 (see also connections)
 (see also PersistenceManagerFactory)
 ConnectionFactory2, 209, 399
 (see also connections)
 (see also PersistenceManagerFactory)
 ConnectionFactory2Name, 209, 400
 (see also connections)
 (see also PersistenceManagerFactory)
 ConnectionFactory2Properties, 401
 ConnectionFactoryMode, 400, 426
 ConnectionFactoryName, 209, 400, 426
 (see also connections)
 (see also PersistenceManagerFactory)
 ConnectionFactoryProperties, 400, 423
 ConnectionPassword, 208, 401, 423
 (see also connections)
 (see also PersistenceManagerFactory)
 ConnectionProperties, 401, 423
 ConnectionRetainMode, 403, 438
 connections, 423
 (see also DataSource)
 accessing DataSource, 427
 configuration, 208
 Connection2DriverName, 399
 Connection2Password, 401
 Connection2Properties, 401

- Connection2URL, 402
- Connection2UserName, 402
- ConnectionDecorators, 412
- ConnectionDriverName, 208, 399
- ConnectionFactory, 209, 399
- ConnectionFactory2, 209, 399
- ConnectionFactory2Name, 209, 400
- ConnectionFactory2Properties, 401
- ConnectionFactoryMode, 400
- ConnectionFactoryName, 209, 400
- ConnectionFactoryProperties, 400
- ConnectionPassword, 208, 401
- ConnectionProperties, 401
- ConnectionRetainMode, 403
- ConnectionURL, 208, 402
- ConnectionUserName, 208, 402
- InitializationSQL, 431
- obtaining from PersistenceManager, 224
- pooling
 - ClosePoolSQL, 424
 - ExceptionAction, 423
 - MaxActive, 423
 - MaxIdle, 423
 - MaxTotal, 423
 - MaxWait, 424
 - MinEvictableIdleTimeMillis, 424
 - RollbackOnReturn, 424
 - TestOnBorrow, 424
 - TestOnReturn, 424
 - TestWhileIdle, 424
 - TimeBetweenEvictionRunsMillis, 424
 - TrackParameters, 424
 - ValidationSQL, 424
 - ValidationTimeout, 424
 - WarningAction, 425
 - WhenExhaustedAction, 425
- usage, 438
- ValidationSQL, 431
- ConnectionURL, 208, 402, 423
 - (see also connections)
 - (see also PersistenceManagerFactory)
- ConnectionUserName, 208, 402, 423
 - (see also connections)
 - (see also PersistenceManagerFactory)
- consistency, 79, 226
 - (see also transactions)
- constrained variables (see variables)
- constructor
 - no-arg constructor requirement, 19, 180
- Container Managed Transactions (see CMT)
- ContainerTable, 512
 - (see also mapping metadata)
- contains, 76
 - (see also EntityManager)
- count, 244
 - (see also aggregates)
- CreateLifecycleListener, 186
 - (see also lifecycle callbacks)
- CURRENT_DATE function, 88
- CURRENT_TIME function, 88
- CURRENT_TIMESTAMP function, 88
- custom mapping, 522
 - field mapping, 523
 - configuration, 524
 - field strategy, 523
 - value handler, 523
- D**
- DataCache, 403
- DataCacheTimeout, 403
- DataSource, 209
 - Kodo, 423
 - managed, 426
 - third party, 423
 - XA, 426
- datastore, 348
- DataStoreCache, 177, 257
 - (see also caching)
 - accessing, 213
 - evict, 257
 - Kodo extensions (see KodoDataStoreCache) (see QueryResult-Cache)
 - pin, 258
- DataStoreIdColumn, 508
 - (see also mapping metadata)
- DatastoreIdentity, 212
 - (see also datastore identity)
 - (see also supported options set)
- datastore identity, 187, 187, 195
 - (see also identity)
 - autoassign strategy, 458, 458
 - mapping, 266, 508
- datastore transactions (see transactions, datastore)
- data transfer object, 72, 219
- DB2, 429
- DBDictionary, 412, 428
- DDL
 - ArrayTypeNames, 433
 - AutoAssignTypeName, 435
 - BigintTypeName, 433
 - BinaryTypeName, 433
 - BitTypeName, 433
 - BlobTypeName, 433
 - CharacterColumnSize, 433
 - CharTypeName, 433
 - ClobTypeName, 433
 - ConstraintNameMode, 430
 - CreatePrimaryKeys, 430
 - DateTypeName, 433
 - DecimalTypeName, 433
 - DistinctTypeName, 433
 - DoubleTypeName, 433
 - FloatTypeName, 433
 - IntegerTypeName, 433
 - JavaObjectTypeNames, 433
 - LongVarbinaryTypeName, 433
 - LongVarcharTypeName, 433
 - MaxAutoAssignNameLength, 430

- build time, 453
- runtime
 - in an EJB container, 454
 - outside a container, 454
- serialization
 - of enhanced types, 455
- Enterprise Java Beans (see EJB)
- entities
 - inheritance, 129
 - (see also inheritance)
 - mapping to database (see mapping metadata)
- entity, 15
 - callback methods, 24
- Entity, 18
 - (see also persistent classes)
 - annotation, 29
- entity identity (see identity)
- EntityListeners, 31
- entity-listeners, 31
- EntityManager, 15, 70
 - as Query factory, 77
 - (see also Query)
 - cache, 76
 - clear, 77
 - closing, 78
 - contains, 76
 - find, 76
 - (see also identity)
 - flush, 76
 - FlushMode, 77
 - getReference, 76
 - (see also identity)
 - Kodo extensions (see KodoEntityManager)
 - lifecycle operations, 71
 - lock, 73
 - merge, 72
 - (see also detachment)
 - obtaining, 66
 - (see also EntityManagerFactory)
 - obtaining the Transaction, 70
 - (see also transactions)
 - persist, 71
 - refresh, 72
 - remove, 71
- EntityManagerFactory, 15, 66
 - closing, 69
 - construction, 63, 66
 - Kodo extensions (see KodoEntityManagerFactory)
 - obtaining EntityManagers, 66
- EntityNotFoundException, 76
- EntityTransaction, 15
- Enumerated, 143
- events
 - lifecycle, 185
 - (see also lifecycle callbacks)
 - remote (see remote, events)
- evict, 222
 - (see also PersistenceManager)
- exceptions

- EJB, 16
- failed object, 178
- JDO, 177
 - nested exceptions, 178
- eXtensible Markup Language (see XML)
- extensions
 - metadata, 194
 - (see also metadata)
- Extent, 176, 230, 540
 - close, 230
 - closeAll, 230
 - embedded objects, 197
 - getCandidateClass, 230
 - getFetchPlan, 230
 - (see also FetchPlan)
 - hasSubclasses, 230
 - iterator, 230
 - Kodo extensions (see KodoExtent)
 - obtaining, 223
 - requires-extent attribute, 195
- externalization, 464
 - external values, 468
 - queries, 467

F

- FetchBatchSize, 404, 441
- FetchDirection, 413, 441
- fetch groups, 469
 - custom configuration, 470
 - default fetch group, 197
 - eager fetching, 473
 - (see also eager fetching)
 - FetchGroups, 404, 470
 - FetchPlan, 254
 - single fields, 471
- FetchPlan, 176, 254, 541
 - (see also fetch groups)
 - detachment options, 255
 - fetch size, 255
 - Kodo extensions (see KodoFetchPlan)
 - obtaining from PersistenceManager, 216
- FetchType, 36
 - (see also eager fetching)
- field
 - persistent (see persistent fields)
- filter, 233
 - (see also Query)
- FilterListeners, 404
- find, 76
 - (see also EntityManager)
- flat, 131, 279
 - (see also inheritance)
- flush, 76, 201, 222
 - (see also EntityManager)
 - (see also PersistenceManager)
 - automatic, 438
 - FlushBeforeQueries, 405
 - FlushBeforeQueries, 405, 438
 - FlushMode, 77

foreign key, 269
 foreign keys, 309, 517
 cascade, 309
 default, 309
 deferred, 310
 delete action, 309
 null, 309
 OrphanedKeyAction, 408
 restrict, 309
 SupportsCascadeDeleteAction, 430
 SupportsDefaultDeleteAction, 430
 SupportsDeferredConstraints, 430
 SupportsForeignKeys, 430
 SupportsNullDeleteAction, 430
 SupportsRestrictDeleteAction, 430
 update action, 309
 forward mapping, 489
 automatic runtime mapping, 495
 hints, 494
 FoxPro, 429

G

GeneratedValue, 34
 generators
 class-table, 556
 mapping metadata, 126
 native, 556
 Seq interface, 555
 SequenceGenerator, 126
 table, 556
 TableGenerator, 127
 time, 557
 value-table, 556
 GetDataStoreConnection, 212
 (see also supported options set)
 getEntityManagerFactory, 63
 (see also Persistence)
 getObjectById, 221, 222
 (see also PersistenceManager)
 getObjectId, 186, 201, 221
 (see also JDOHelper)
 (see also PersistenceManager)
 getObjectIdClass, 221
 (see also PersistenceManager)
 getObjectsById, 222
 (see also PersistenceManager)
 getPersistenceManager, 202
 (see also JDOHelper)
 getPersistenceManagerFactory, 205
 (see also JDOHelper)
 getReference, 76
 (see also EntityManager)
 getVersion, 201
 (see also JDOHelper)
 grouping, 245
 (see also aggregates)

H

having, 245
 (see also aggregates)
 hollow, 203, 221
 (see also lifecycle states)
 horizontal, 274
 (see also inheritance)
 Hypersonic SQL, 429

I

id
 fields, 21
 (see also persistent fields)
 Id, 34, 123
 IdClass, 30
 identity, 455
 application, 187, 187
 application identity tool, 456
 class requirements, 187
 hierarchy, 190
 class requirements, 22
 creating identity objects, 221
 datastore, 187, 187, 455, 481
 determining identity class, 221
 hierarchy, 23
 identity object, 186
 JDO, 186
 JPA, 21
 mapping, 123
 numeric, 21, 186
 qualitative, 21, 186
 retrieving from a persistent object, 201, 221
 retrieving objects by identity, 76, 76, 221, 222, 222
 single field, 187, 190
 specifying in metadata, 195
 uniqueness requirement, 22, 186
 identity class, 22
 (see also identity)
 identity fields, 19
 (see also persistent fields)
 IgnoreCache, 210, 216
 (see also PersistenceManager)
 (see also PersistenceManagerFactory)
 IgnoreChanges, 405
 immutable
 persistent field types, 20, 181
 impedance mismatch, 129, 273
 implicit parameters (see parameters)
 implicit variables (see variables)
 indexes, 311, 517
 MaxIndexesPerTable, 430
 MaxIndexNameLength, 430
 unique, 311
 IN expressions, 85
 Informix, 429
 inheritance
 discriminator, 138, 281
 (see also discriminator)
 distributed, 274
 flat, 131, 279

- horizontal, 274
- JOINED strategy, 131
 - advantages, 133
 - disadvantages, 133
- mapping, 129, 273
- new-table strategy, 275
 - advantages, 277, 278
 - disadvantages, 277, 278
- of persistent classes, 20, 180
- SINGLE_TABLE strategy, 130
 - advantages, 131
 - disadvantages, 131
- SubclassFetchMode, 415
- subclass-table strategy, 273
 - abstract classes, 274
 - advantages, 274
 - application identity, 274
 - considerations, 274
 - disadvantages, 274
 - relations, 274
- superclass-table strategy, 279
 - advantages, 279
 - disadvantages, 279
- TABLE_PER_CLASS strategy, 134
 - advantages, 135
 - disadvantages, 135
- vertical, 131, 275
- InstanceCallbacks, 183
 - (see also lifecycle callbacks)
- jdoPostLoad, 183
- jdoPreClear, 184
- jdoPreDelete, 184
- jdoPreStore, 183
- InstanceLifecycleEvent, 185, 185
 - (see also lifecycle callbacks)
- InstanceLifecycleListener, 185, 185
 - (see also lifecycle callbacks)
 - adding to PersistenceManager, 216
 - (see also PersistenceManager)
 - adding to PersistenceManagerFactory, 211
 - (see also PersistenceManagerFactory)
- ClearLifecycleListener, 186
- CreateLifecycleListener, 186
- DeleteLifecycleListener, 186
- LoadLifecycleListener, 185
 - removing from PersistenceManager, 216
 - (see also PersistenceManager)
 - removing from PersistenceManagerFactory, 211
 - (see also PersistenceManagerFactory)
- StoreLifecycleListener, 186
- interfaces
 - as persistent field types, 182
 - (see also persistent fields)
- IntIdentity, 192
 - (see also single field identity)
- InverseManager, 405, 459
- isDirty, 202
 - (see also dirty)
 - (see also JDOHelper)
- IS EMPTY expressions, 85
- isNew, 202
 - (see also JDOHelper)
- IS NULL expressions, 85
- isolation, 79, 226
 - (see also transactions)
- isPersistent, 202
 - (see also JDOHelper)
 - (see also persistent objects)
- isTransactional, 202
 - (see also JDOHelper)
 - (see also transactional)
- J**
- JavaBean
 - as query result class, 246
 - (see also Query)
- Java Connector Architecture (see JCA)
- Java Database Connectivity (see JDBC)
- Java Data Objects (see JDO)
- Java Data Objects Query Language (see JDOQL)
- Java Naming and Directory Interface (see JNDI)
- Java Web Start applications, 582
- JBoss, 528, 530, 530, 531
- JCA, 66, 207, 209
 - Borland 5.2-6, 532
 - deployment, 527, 528
 - JBoss 3.0, 530
 - JBoss 3.2.x, 530
 - JBoss 4.x, 528, 531
 - JRun 4, 532
 - Sun JES 8-8.1, 531
 - SunONE 7, 531
 - Weblogic 6.1-7.x, 528
 - Weblogic 8.1, 529
 - Weblogic 9, 527, 530
 - Websphere 5, 531
- JDataStore, 429
- JDBC, 13, 174, 423
 - accessing DataSource, 427
 - connection access (see connections)
 - DBDictionary, 412
 - DriverVendor, 430
 - JDBCListeners, 413
 - large result sets (see large result sets)
 - log messages, 417
 - QueryTimeout, 425
 - SupportsQueryTimeout, 432
 - TransactionIsolation, 415
 - transaction isolation, 436
 - UpdateManager, 416
- JDBCListeners, 413
- JDO, 14, 173, 175
 - architecture, 176
 - configuration, 397
 - enhancer, 179
 - (see also enhancer)
 - exceptions, 177
 - (see also exceptions)

- for relational databases (see JDOR)
 - identity, 186
 - (see also identity)
 - identity object, 186
 - (see also identity)
 - metadata, 193
 - (see also metadata)
 - query language (see JDOQL)
 - vs EJB 2, 175
 - vs JDBC, 174
 - vs JPA, 175
 - vs ODBs, 175
 - vs ORM products, 174
 - vs serialization, 174
 - why, 174
- jdoc, 347
- JDOCanRetryException, 177
 - (see also exceptions)
 - JDODataStoreException, 177
 - (see also exceptions)
 - JDOException, 177
 - (see also exceptions)
 - JDOCanRetryException, 177
 - JDODataStoreException, 177
 - JDOFatalDataStoreException, 177
 - JDOFatalException, 177
 - JDOFatalInternalException, 177
 - JDOFatalUserException, 177
 - JDOObjectNotFoundException, 177
 - JDOOptimisticVerificationException, 177
 - JDOUnsupportedOptionException, 177
 - JDOUserException, 177
 - JDOFatalDataStoreException, 177
 - (see also exceptions)
 - JDOFatalException, 177
 - (see also exceptions)
 - JDOFatalInternalException, 177
 - (see also exceptions)
 - JDOFatalUserException, 177
 - (see also exceptions)
 - JDOHelper, 176, 201
 - getObjectId, 186, 201, 221
 - (see also identity)
 - getPersistenceManager, 202
 - (see also PersistentManager)
 - getPersistenceManagerFactory, 205, 394
 - getVersion, 201
 - (see also version)
 - isDirty, 202
 - isNew, 202
 - isPersistent, 202
 - isTransactional, 202
 - lifecycle operations, 202
 - (see also lifecycle states)
 - makeDirty, 182, 201
 - (see also dirty)
 - persistence capable operations, 201
 - JDOObjectNotFoundException, 177, 221
 - (see also exceptions)
 - JDOOptimisticVerificationException, 177, 223
 - (see also exceptions)
 - JDOPermission
 - closePersistenceManagerFactory, 213
 - jdoPostLoad, 183
 - (see also InstanceCallbacks)
 - jdoPreClear, 184
 - (see also InstanceCallbacks)
 - jdoPreDelete, 184
 - (see also InstanceCallbacks)
 - jdoPreStore, 183
 - (see also InstanceCallbacks)
 - JDOQL, 176, 213, 234
 - (see also Query)
 - (see also supported options set)
 - advanced, 236
 - aggregate extension, 552
 - configuration, 552
 - aggregates, 244
 - (see also aggregates)
 - delete by query, 252
 - differences from Java, 234
 - distinct, 243
 - grouping, 245
 - (see also aggregates)
 - having, 245
 - (see also aggregates)
 - language extension, 550
 - configuration, 552
 - custom, 552
 - getColumn, 551
 - sql, 551
 - method support, 234
 - non-distinct results, 552
 - parameters, 236
 - (see also parameters)
 - projections, 241
 - (see also projections)
 - regular expression support, 234
 - single-string, 248
 - creating single-string queries, 249
 - grammar, 249
 - subqueries, 553
 - subselects
 - RequiresAliasForSubselect, 432
 - SupportsSubselect, 432
 - variables, 236
 - (see also variables)
 - jdoquery files (see named queries)
 - JDOR, 259
 - object-relational mapping, 260
 - (see also mapping metadata)
 - sequences, 318
 - (see also Sequence)
 - SQL queries, 320
 - (see also SQL queries)
 - JDOUnsupportedOptionException, 177
 - (see also exceptions)
 - JDOUserException, 177

(see also exceptions)
 JMX (see management)
 JNDI, 66, 207, 209, 209
 retrieving bound PersistenceManagerFactory, 205
 joins, 269
 class criteria, 520
 constant, 506
 CrossJoinClause, 431
 forward, 270
 InnerJoinClause, 431
 inverse, 270
 JoinSyntax, 431
 mapping shortcuts, 270
 non-primary key, 506
 non-standard, 506
 outer (see outer joins)
 OuterJoinClause, 431
 partial primary key, 506
 RequiresConditionForCrossJoin, 431
 self joins, 271
 inverse, 272
 source table, 270
 syntax options, 436
 target table, 270
 join table, 156
 JPA, 14
 identity, 21
 (see also identity)
 metadata, 28
 (see also metadata)
 XML, 42
 (see also metadata)
 vs EJB 2, 14
 vs JDBC, 13
 vs JDO, 14
 vs ODBs, 14
 vs ORM products, 13
 vs serialization, 13
 why, 13
 JPQL, 15
 aggregate extension, 552
 configuration, 552
 language extension, 550
 configuration, 552
 custom, 552
 getColumn, 551
 sql, 551
 subselects
 RequiresAliasForSubselect, 432
 SupportsSubselect, 432
 JP Query, 82
 (see also JPQL)
 JRun, 532

K

KeyColumn, 515
 (see also mapping metadata)
 KeyEmbeddedMapping, 516
 (see also mapping metadata)

KeyJoinColumn, 516
 (see also mapping metadata)
 kodo.properties, 393
 kodo.xml, 393
 kodoc, 330 (see enhancer)
 KodoDataStoreCache, 542
 KodoEntityManager, 540
 extending, 539
 KodoEntityManagerFactory, 540
 KodoExtent, 542
 KodoFetchPlan, 542
 KodoJDOHelper, 542
 Kodo JPA/JDO, 3
 enterprise edition, 628
 performance pack, 628
 standard edition, 628
 KodoPersistence, 541
 KodoPersistenceManager, 541
 extending, 539
 KodoPersistenceManagerFactory, 541
 KodoQuery, 540, 542

L

large result sets, 440
 FetchBatchSize, 404
 FetchDirection, 413
 fields, 461
 interaction with eager fetching, 475
 LRSSize, 413
 ResultSetType, 414
 lazy loading, 203, 473
 (see also eager fetching)
 (see also fetch groups)
 locking behavior, 547
 LENGTH function, 87
 LicenseKey, 405
 lifecycle callbacks, 24, 183, 202
 (see also lifecycle states)
 callback methods, 24
 event framework, 183
 (see also InstanceLifecycleEvent)
 (see also InstanceLifecycleListener)
 InstanceCallbacks, 183
 LifecycleListener
 DirtyLifecycleListener, 186
 lifecycle listeners
 hierarchy, 26
 lifecycle states, 202
 calculating with JDOHelper, 204
 (see also JDOHelper)
 hollow, 203
 persistent-clean, 202
 persistent-deleted, 203
 persistent-dirty, 202
 (see also dirty)
 persistent-new, 202
 persistent-new-deleted, 202
 persistent-nontransactional, 203
 transient, 202

- transient-clean, 203
 - transient-dirty, 203
 - (see also dirty)
 - transient-transactional, 203
 - lightweight persistence, 12
 - LIKE expressions, 84
 - LinkedList, 212
 - (see also persistent fields)
 - (see also supported options set)
 - List, 212
 - (see also persistent fields)
 - (see also supported options set)
 - LoadCallback (see InstanceCallbacks)
 - LoadLifecycleListener, 185
 - (see also lifecycle callbacks)
 - LOB, 143
 - LOCATE function, 87
 - lock groups, 476
 - (see also locking)
 - mapping metadata, 479
 - subclasses, 477
 - locking, 73, 542
 - (see also EntityManager)
 - behavior, 547
 - caveats, 547
 - defaults configuration, 542
 - ForUpdateClause, 431
 - levels, 542
 - lock groups, 476
 - LockManager, 406, 546
 - LockTimeout, 406
 - ReadLockLevel, 409
 - runtime APIs, 544
 - runtime configuration, 543
 - SimulateLocking, 432
 - SupportsLockingWithDistinctClause, 431
 - SupportsLockingWithInnerJoin, 431
 - SupportsLockingWithMultipleTables, 432
 - SupportsLockingWithOrderClause, 432
 - SupportsLockingWithOuterJoin, 431
 - SupportsLockingWithSelectRange, 432
 - SupportsSelectForUpdate, 431
 - TableForUpdateClause, 431
 - timeout, 542
 - WriteLockLevel, 411
 - LockManager, 406, 546
 - LockTimeout, 406, 542
 - Log, 406, 417
 - logging, 417
 - Apache Commons, 420
 - channels, 417
 - custom, 421
 - default, 418
 - disabling, 419
 - JDK 1.4, 420
 - Log, 406
 - Log4j, 420
 - LongIdentity, 192
 - (see also single field identity)
 - LOWER function, 87
 - LRSSize, 413, 441
- ## M
- makeDirty, 182, 201
 - (see also JDOHelper)
 - makeNontransactional, 217
 - (see also PersistenceManager)
 - makePersistent, 202, 202, 217
 - (see also PersistenceManager)
 - makeTransactional, 203, 217
 - (see also PersistenceManager)
 - makeTransient, 217
 - (see also PersistenceManager)
 - ManagedRuntime, 406, 535
 - managed transactions (see transactions)
 - management, 591
 - configuring logging, 595
 - configuring remote, 594, 594
 - configuring WebLogic 8.1, 595
 - kodo.ManagementConfiguration, 593
 - log messages, 418
 - ManagementConfiguration, 407
 - ManagementConfiguration, 407
 - Management Console, 595
 - many-many, 156, 298
 - (see also persistent fields)
 - many-one, 153, 294
 - (see also persistent fields)
 - ManyToMany, 40
 - ManyToOne, 36
 - map
 - metadata, 198
 - (see also persistent fields)
 - MapKey, 41
 - mappedBy, 39
 - (see also mapping metadata)
 - mapped-by, 296, 301, 303
 - (see also mapping metadata)
 - MappedSuperclass, 30
 - Mapping, 211, 260, 407
 - (see also PersistenceManagerFactory)
 - MappingDefaults, 413, 500
 - MappingFactory, 414, 503
 - import/export mapping data, 505
 - mapping metadata, 39, 118, 260, 489
 - (see also mappedBy property)
 - association table collection fields, 156, 298
 - (see also persistent fields)
 - autoassign strategy, 266
 - automatic runtime mapping, 495
 - basic collection fields, 297
 - (see also persistent fields)
 - basic fields, 143, 288
 - (see also persistent fields)
 - BLOB fields, 291
 - (see also persistent fields)
 - class
 - table attribute, 119, 264

- CLOB fields, 290
 - (see also persistent fields)
- collections, 512
 - JPA one-sided one-many, 514
- Column, 123
 - columnDefinition property, 123
 - insertable property, 123
 - length property, 123
 - name property, 123
 - nullable property, 123
 - precision property, 123
 - scale property, 123
 - table property, 123
 - updatable property, 123
- column, 268, 510
 - allows-null attribute, 269
 - default-value attribute, 269
 - jdbc-type attribute, 268
 - (see also SQL)
 - length attribute, 269
 - name attribute, 268
 - scale attribute, 269
 - sql-type attribute, 268
 - (see also SQL)
 - target attribute, 269, 269, 272
 - target-field attribute, 269
- custom mapping (see custom mapping)
- datastore identity, 266, 508
 - (see also identity)
 - column attribute, 267
 - identity strategy, 266
 - increment strategy, 266
 - native strategy, 266
 - sequence attribute, 266
 - (see also Sequence)
 - strategy attribute, 266
- defaults (see MappingDefaults)
- delete-action attribute, 309
- direct relation fields, 153, 294
 - (see also persistent fields)
- discriminator, 138, 281
 - (see also discriminator)
 - column attribute, 281
 - strategy attribute, 281
 - value attribute, 281
- DTD, 261
- embedded fields, 150, 306
 - (see also embedded)
 - null-indicator-column attribute, 307
- enums, 143
- extensions
 - class criteria, 520
 - (see also joins)
 - discriminator strategy, 522
 - (see also custom mapping)
 - eager fetch mode, 520
 - (see also eager fetching)
 - insertable, 522
 - lock-group, 522
 - nonpolymorphic, 520
 - strategy, 522, 522
 - (see also custom mapping)
 - subclass fetch mode, 520
 - (see also eager fetching)
 - updatable, 522
 - version strategy, 522
 - (see also custom mapping)
- field
 - column attribute, 288
 - join element, 292
 - key element, 304
 - mapped-by attribute, 296, 301, 303
 - sequence attribute, 292
 - serialized attribute, 291
 - table attribute, 292
 - value element, 304
 - value-strategy attribute, 292
- field mapping, 143, 287
 - (see also persistent fields)
- foreign keys, 309, 517
 - (see also foreign keys)
 - deferred attribute, 310
 - delete-action attribute, 310
 - name attribute, 310
 - update-action attribute, 310
- forward mapping (see forward mapping)
- generators, 126
 - (see also SequenceGenerator)
 - (see also TableGenerator)
- identity, 123
- indexed attribute, 311
- indexes, 311, 517
 - (see also indexes)
 - name attribute, 312
 - unique attribute, 312
- inheritance, 129, 273
 - (see also inheritance)
 - JOINED strategy, 131
 - new-table strategy, 275
 - SINGLE_TABLE strategy, 130
 - strategy attribute, 130, 273
 - subclass-table strategy, 273
 - superclass-table strategy, 279
 - TABLE_PER_CLASS strategy, 134
- inverse key collection fields, 301
 - (see also persistent fields)
- joins, 269
 - (see also joins)
- JPA additions, 508
- limitations, 519
 - table-per-class, 519
- loading and storing (see MappingFactory)
- LOB types, 143
- map fields, 158, 304
 - (see also persistent fields)
- MappingDefaults, 413
- MappingFactory, 414
- maps, 515

- meet-in-the-middle mapping (see meet-in-the-middle mapping)
- multi-column mappings, 510
- placement, 260
- reverse mapping (see reverse mapping)
- secondary table fields, 149, 292
 - (see also persistent fields)
- sequences, 263
 - (see also Sequence)
- sequence strategy, 266
- SynchronizeMappings, 415
- temporal types, 144
- unique attribute, 312
- unique constraints, 122, 312, 518
 - (see also unique constraints)
 - deferred attribute, 312
 - name attribute, 312
- uuid-hex, 35
- uuid-hex strategy, 266
- uuid-string, 35
- uuid-string strategy, 266
- version, 284, 509
 - (see also version)
 - column attribute, 285
 - lock group mapping, 479
 - (see also lock groups)
 - strategy attribute, 285
- mappingtool, 330, 333, 347, 348
- mapping tool, 489
 - (see also forward mapping)
 - Ant task, 610
 - DDL generation, 493
 - use cases, 491
- Maven, 611
- max, 244
 - (see also aggregates)
- MBean, 591, 598
 - data cache, 600
 - prepared statement, 600
 - query cache, 600
 - TimeWatch, 600
- Mbean, 600
 - datasource, 600
 - log, 600
 - runtime, 601
- meet-in-the-middle mapping, 495
- merge, 72
 - (see also detachment)
- metadata, 28, 193
 - array, 197
 - dependent-element attribute, 197
 - (see also dependent)
 - embedded-element attribute, 198
 - (see also embedded)
 - Basic, 35
 - CascadeType, 37
 - class
 - detachable attribute, 196
 - (see also detachment)
 - embedded-only attribute, 195
 - (see also embedded)
 - identity-type attribute, 195
 - (see also identity)
 - objectid-class attribute, 195
 - (see also identity)
 - persistence-modifier attribute, 195
 - (see also persistent classes)
 - requires-extent attribute, 195
 - (see also Extent)
 - class names, 195
 - collection
 - element-type attribute, 198
 - (see also persistent fields)
 - DTD, 193
 - Embeddable, 30
 - Embedded, 36
 - EmbeddedId, 35
 - Entity, 29
 - EntityListeners, 31
 - extensions, 194, 483
 - data cache, 483
 - (see also caching)
 - dependent, 483
 - detached state field, 484
 - (see also detachment)
 - externalizer, 487
 - (see also externalization)
 - external values, 487
 - (see also externalization)
 - factory, 487
 - (see also externalization)
 - fetch groups, 483
 - (see also fetch groups)
 - inverse-logical, 486
 - (see also bidirectional relations)
 - key attribute, 194
 - lock group, 486
 - (see also locking)
 - lrs, 485
 - (see also large result sets)
 - order-by, 485
 - read-only, 486
 - (see also persistent fields)
 - type, 486
 - (see also persistent fields)
 - value attribute, 194
 - vendor-name attribute, 194
 - FetchType, 36
 - field
 - default-fetch-group attribute, 197
 - (see also fetch groups)
 - dependent attribute, 197
 - (see also dependent)
 - embedded attribute, 197
 - (see also embedded)
 - null-value attribute, 197, 269
 - (see also persistent fields)
 - persistence-modifier attribute, 196
 - (see also persistent fields)

- primary-key attribute, 197
 - (see also identity)
 - (see also persistent fields)
 - GeneratedValue, 34
 - generating default metadata, 480
 - Id, 34
 - IdClass, 30
 - JPA additions, 481
 - loading and storing (see `MetaDataFactory`)
 - log messages, 417
 - ManyToMany, 40
 - ManyToOne, 36
 - map, 198
 - dependent-key attribute, 198
 - (see also dependent)
 - dependent-value attribute, 198
 - (see also dependent)
 - embedded-key attribute, 198
 - (see also embedded)
 - embedded-value attribute, 198
 - (see also embedded)
 - key-type attribute, 198
 - (see also persistent fields)
 - value-type attribute, 198
 - (see also persistent fields)
 - MapKey, 41
 - MappedSuperclass, 30
 - mapping metadata (see mapping metadata)
 - `MetaDataFactory`, 407
 - OneToMany, 38
 - OneToOne, 39
 - OrderBy, 41
 - placement, 200
 - property access, 33
 - query metadata (see named queries)
 - `RetryClassRegistration`, 410
 - Transient, 34
 - Version, 35
 - XSD, 42
 - `MetaDataFactory`, 407
 - `MethodQL`, 554
 - Microsoft Access, 429
 - migration, 648
 - from Kodo 3 to Kodo 4, 648
 - min, 244
 - (see also aggregates)
 - MOD function, 88
 - monitoring (see management)
 - Multithreaded, 209, 216, 407
 - (see also `PersistenceManager`)
 - (see also `PersistenceManagerFactory`)
 - (see also threading)
 - mutable
 - persistent field types, 20, 181
 - (see also persistent fields)
 - (see also proxies)
 - MySQL, 429, 435
 - (see also `DBDictionary`)
 - `DriverDeserializesBlobs`, 435
 - `TableType`, 435
 - `UseClobs`, 435
- ## N
- named queries, 250
 - (see also `Query`)
 - constructing, 252
 - defining, 250
 - executing, 252
 - SQL, 324
 - Native
 - queries (see SQL queries)
 - `newObjectIdInstance`, 221
 - (see also `PersistenceManager`)
 - `NonDurableIdentity`, 212
 - (see also identity)
 - (see also supported options set)
 - `NontransactionalRead`, 203, 210, 212, 228, 408
 - (see also `PersistenceManagerFactory`)
 - (see also `Transaction`)
 - (see also supported options set)
 - (see also `Transaction`)
 - `NontransactionalWrite`, 203, 210, 212, 212, 228, 408
 - (see also `PersistenceManagerFactory`)
 - (see also `Transaction`)
 - (see also supported options set)
 - (see also `Transaction`)
 - normalized, 133, 277
 - NOT expressions, 85
 - `NullCollection`, 212
 - (see also supported options set)
 - numeric identity, 21, 186
 - (see also identity)
- ## O
- Object
 - as persistent field type, 21, 182
 - (see also persistent fields)
 - object database (see ODB)
 - object filtering, 232
 - (see also `Query`)
 - advanced, 236
 - object identity (see identity)
 - object-relational mapping (see ORM)
 - ODB, 14, 175
 - ODBMG, 14, 175
 - offline (see remote)
 - one-many, 156, 298, 301
 - (see also persistent fields)
 - one-one, 153, 294
 - (see also persistent fields)
 - OneToMany, 38
 - OneToOne, 39
 - Optimistic, 203, 210, 212, 228, 408
 - (see also `PersistenceManagerFactory`)
 - (see also `Transaction`)
 - (see also transactions)
 - (see also supported options set)

(see also Transaction)
 optimistic transactions (see transactions, optimistic)
 optimization guidelines, 612
 Oracle, 429, 435
 (see also DBDictionary)
 AutoAssignSequenceName, 435
 MaxEmbeddedBlobSize, 435
 MaxEmbeddedClobSize, 435
 TestBatchSupport, 436
 UseTriggersForAutoAssign, 435
 OrderBy, 41
 OrderColumn, 513
 (see also mapping metadata)
 ordering (see Query, ordering)
 ORM, 13, 118, 174, 260
 (see also mapping metadata)
 OrphanedKeyAction, 408
 outer joins, 282, 293, 473

P

parameters, 236
 (see also JDOQL)
 implicit, 236
 in SQL queries, 117, 322
 (see also SQL queries)
 query by example, 238
 PCClasses, 452
 permissions (see JDOPermission)
 persist, 71
 (see also EntityManager)
 Persistence, 15, 63
 createEntityManagerFactory, 394
 getEntityManagerFactory, 63
 persistence aware, 180
 specifying in metadata, 195
 PersistenceCapable, 176, 179
 (see also persistent classes)
 vs persistence aware, 180
 persistence capable (see Entity) (see PersistenceCapable)
 persistence context, 67
 PersistenceContextType (see persistence context)
 PersistenceManager, 176, 214
 adding and removing InstanceLifecycleListeners , 216
 (see also lifecycle callbacks)
 as Extent factory, 223
 (see also Extent)
 as Query factory, 223
 (see also Query)
 as Sequence factory, 224
 (see also Sequence)
 attachCopy, 219
 (see also detachment)
 cache, 222
 checkConsistency, 223
 closing, 225
 defaults, 209
 deletePersistent, 217
 detachCopy, 219
 (see also detachment)

evict, 222
 flush, 222
 getObjectById, 221, 222
 (see also identity)
 getObjectId, 221
 (see also identity)
 (see also JDOHelper)
 getObjectIdClass, 221
 (see also identity)
 getObjectsById, 222
 (see also identity)
 IgnoreCache, 216
 in a managed environment, 225
 Kodo extensions (see KodoPersistenceManager)
 lifecycle operations, 216
 makeNontransactional, 217
 makePersistent, 217
 makeTransactional, 217
 makeTransient, 217
 Multithreaded, 216
 newNamedQuery, 252
 newObjectIdInstance, 221
 (see also identity)
 obtaining, 211
 (see also PersistenceManagerFactory)
 obtaining datastore connection, 224
 (see also connections)
 obtaining the FetchPlan, 216
 (see also FetchPlan)
 obtaining the Transaction, 216
 (see also transactions)
 refresh, 218
 retrieve, 218
 retrieving from a managed object, 202
 user objects, 215
 PersistenceManagerFactory, 176, 207
 adding and removing InstanceLifecycleListeners , 211
 (see also lifecycle callbacks)
 closing, 213
 connection configuration, 208
 (see also connections)
 ConnectionDriverName, 208
 ConnectionFactory, 209
 ConnectionFactory2, 209
 ConnectionFactory2Name, 209
 ConnectionFactoryName, 209
 ConnectionPassword, 208
 ConnectionURL, 208
 ConnectionUserName, 208
 construction, 205, 207
 DetachAllOnCommit, 211
 IgnoreCache, 210
 Kodo extensions (see KodoPersistenceManagerFactory)
 Mapping, 211
 Multithreaded, 209
 NontransactionalRead, 210
 NontransactionalWrite, 210
 obtaining PersistenceManagers, 211
 Optimistic, 210

- PersistenceManager defaults, 209
 - (see also PersistenceManager)
- PersistenceManagerFactoryClass, 397
- properties, 212
- RestoreValues, 210
- RetainValues, 210
- Transaction defaults, 209
 - (see also Transaction)
- PersistenceManagerfactory
 - supported options, 211
- PersistenceManagerFactoryClass, 397
- PersistenceServer, 408
- persistent classes, 18, 179, 202, 452
 - (see also persistent objects)
 - detachable, 196
 - (see also detachment)
 - embedded-only, 195
 - (see also embedded)
 - field restrictions, 20, 181
 - (see also persistent fields)
 - inheritance, 273
 - (see also inheritance)
 - inheritance of, 20, 180
 - (see also inheritance)
 - JPA id requirement, 19
 - JPA version requirement, 19
 - lifecycle callbacks, 24, 183
 - (see also lifecycle callbacks)
 - list, 452
 - mapping to database (see mapping metadata)
 - no-arg constructor requirement, 19, 180
 - property access, 33
 - restrictions on, 19, 180
- persistent-clean, 202
 - (see also lifecycle states)
- persistent data, 12, 173
- persistent-deleted, 203
 - (see also lifecycle states)
- persistent-dirty, 202
 - (see also lifecycle states)
- persistent fields, 460, 482 (see eager fetching)
 - array metadata, 197
 - arrays, 182
 - association table collections, 298
 - bidirectional, 301
 - (see also bidirectional relations)
 - autoassign strategy, 292, 458
 - automatic field values, 291
 - AutoAssignClause, 435
 - AutoAssignSequenceName, 435
 - AutoAssignTypeName, 435
 - LastGeneratedKeyQuery, 435
 - UseTriggersForAutoAssign, 435
 - basic, 143, 288
 - basic collections, 297
 - BLOB mapping, 291
 - calendar, 461
 - checking for ArrayList support, 212
 - checking for array support, 212
 - checking for LinkedList support, 212
 - checking for List support, 212
 - checking for TreeMap support, 212
 - checking for TreeSet support, 212
 - checking for Vector support, 212
 - CLOB mapping, 290
 - collection metadata, 198, 482
 - comparators, 460
 - default persistent types, 196
 - dependent (see dependent)
 - direct relations, 153, 294
 - based on inverse keys, 295
 - bidirectional, 296
 - (see also bidirectional relations)
 - embedded (see embedded)
 - externalization (see externalization)
 - fetch groups (see fetch groups)
 - field rollback, 460
 - id, 21
 - immutable types, 20, 181
 - in secondary tables, 149, 292
 - inverse key collections, 301
 - bidirectional, 303
 - (see also bidirectional relations)
 - join table collections, 156
 - map metadata, 198, 483
 - mapping metadata, 143, 287
 - maps, 158, 304
 - marking as changed, 201
 - mutable types, 20, 181
 - (see also proxies)
 - null value treatment, 197
 - of interface types, 182
 - of unknown types, 21, 182
 - primary key, 187
 - specifying in metadata, 197
 - proxies (see proxies)
 - reading outside a transaction (see NontransactionalRead)
 - read only, 486
 - restrictions on, 20, 181
 - StorageLimitationsFatal, 432
 - StoreCharsAsNumbers, 432
 - StoreLargeNumbersAsStrings, 432
 - superclass field mapping, 288
 - temporal, 144
 - user-defined types, 20, 182
 - writing outside a transaction (see NontransactionalWrite)
- persistent-new, 202
 - (see also lifecycle states)
- persistent-new-deleted, 202
 - (see also lifecycle states)
- persistent-nontransactional, 203
 - (see also lifecycle states)
- persistent objects, 202
 - (see also persistent classes)
 - deleting, 71, 217
 - example, 75, 219
 - evicting, 222
 - identity (see identity)

- lifecycle states (see lifecycle states)
 - making nontransactional, 217
 - making transactional, 217
 - making transient, 217
 - persisting, 71, 217
 - example, 74, 218
 - querying (see Query)
 - refreshing state, 72, 218
 - retrieving state, 218
 - retrieving with SQL, 116, 321
 - (see also SQL queries)
 - updating
 - example, 74, 218
 - persistent properties, 33
 - (see also persistent fields)
 - pessimistic transactions (see transactions, pessimistic)
 - plugins (see configuration)
 - Pointbase, 429
 - PostgreSQL, 429
 - PostLoad, 24
 - (see also lifecycle callbacks)
 - PostPersist, 24
 - (see also lifecycle callbacks)
 - PostRemove, 25
 - (see also lifecycle callbacks)
 - PostUpdate, 25
 - (see also lifecycle callbacks)
 - prepared statement
 - batching, 440
 - BatchLimit, 432
 - BatchParameterLimit, 432
 - SupportsTotalCountsForBatch, 432
 - SupportsUpdateCountsForBatch, 432
 - pooling
 - MaxCachedStatements, 425
 - PrePersist, 24
 - (see also lifecycle callbacks)
 - PreRemove, 25
 - (see also lifecycle callbacks)
 - PreUpdate, 24
 - (see also lifecycle callbacks)
 - primary key, 221, 266, 508
 - fields, 187
 - (see also persistent fields)
 - profiling, 603
 - log messages, 418
 - projections, 241
 - (see also JDOQL)
 - distinct, 243
 - of column data, 322
 - (see also SQL queries)
 - of elements, keys, and values, 243
 - of identity objects, 242
 - of variables, 243
 - result aliases, 247
 - proxies, 461
 - custom, 463
 - large result set, 461
 - ProxyManager, 409, 463
 - smart, 461
 - ProxyManager, 409
- ## Q
- qualitative identity, 21, 186
 - (see also identity)
 - queries (see Query)
 - checking for JDOQL support, 213
 - checking for SQL support, 213
 - Query, 15, 176, 232
 - (see also JDOQL)
 - AggregateListeners, 397
 - aggregates, 244
 - (see also aggregates)
 - by example, 238
 - candidate class, 233
 - candidate objects, 233
 - close, 240
 - closeAll, 240
 - compile, 239
 - creating, 77, 223
 - declareImports, 236
 - declareParameters, 236
 - (see also parameters)
 - declareVariables, 236
 - (see also variables)
 - default result string, 247
 - distinct, 243
 - execute, 240
 - executeWithArray, 240
 - executeWithMap, 240
 - filter, 233
 - FilterListeners, 404
 - FlushBeforeQueries, 405
 - getFetchPlan, 239
 - (see also FetchPlan)
 - grouping, 245
 - (see also aggregates)
 - having, 245
 - (see also aggregates)
 - imports, 236
 - Kodo extensions (see KodoQuery)
 - language extensions, 550
 - (see also JDOQL)
 - (see also JPQL)
 - limits, 240
 - log messages, 417
 - MethodQL (see MethodQL)
 - named (see named queries)
 - object filtering, 232
 - ordering, 241
 - projections, 241
 - (see also projections)
 - result caching, 566
 - result class, 246
 - generic, 248
 - JavaBean, 246
 - Map, 248
 - result range, 240

- SupportsSelectEndIndex, 432
- SupportsSelectStartIndex, 432
- result shape, 245
- setCandidates, 233
- setClass, 233
- setFilter, 233
- setGrouping, 245
- setOrdering, 241
- setRange, 240
- setResult, 242
 - (see also projections)
- setResultClass, 246
- setUnique, 240
- single-string (see JDOQL, single-string)
- SQL (see SQL queries)
- unique result, 240
- QueryCache, 409
- QueryCompilationCache, 409
- query metadata (see named queries)
- QueryResultCache, 541, 542

R

- ReadLockLevel, 409, 542
- refresh, 72, 218
 - (see also EntityManager)
 - (see also PersistenceManager)
- relational database, 259
 - (see also JDOR)
 - accessing multiple databases, 438
 - Kodo support, 428
 - (see also DBDictionary)
- remote, 577, 581
 - client, 583
 - data compression and filtering, 586
 - deployment, 587
 - DynamicDataStructs, 404
 - events, 587
 - common properties, 589
 - configuration, 587
 - customization, 590
 - JMS, 587
 - RemoteCommitListener, 587
 - RemoteCommitProvider, 587
 - TCP, 588
 - HTTP server, 583
 - log messages, 417
 - PersistenceServer, 408
 - RemoteCommitProvider, 410
 - TCP server, 582
- RemoteCommitProvider, 410
- remove, 71
 - (see also EntityManager)
- RestoreState, 410, 460
- RestoreValues, 202, 202, 203, 210, 228
 - (see also PersistenceManagerFactory)
 - (see also Transaction)
 - (see also Transaction)
- ResultSetType, 414, 441
- RetainState, 410

- RetainValues, 203, 210, 228
 - (see also PersistenceManagerFactory)
 - (see also Transaction)
 - (see also Transaction)
- retrieve, 218
 - (see also PersistenceManager)
- RetryClassRegistration, 410
- reverse mapping, 495
- reverse mapping tool, 495, 495
 - (see also reverse mapping)
- Ant task, 610

S

- savepoint, 548
- schema
 - create with schema tool, 446
 - DDL (see DDL)
 - default, 443
 - log messages, 418
 - reflection, 443
 - RequiresAutoCommitForMetaData, 435
 - SchemaCase, 431
 - SupportsNullTableForGetColumns, 434
 - SupportsNullTableForGetImportedKeys, 434
 - SupportsNullTableForGetIndexInfo, 434
 - SupportsNullTableForGetPrimaryKeys, 434
 - SupportsSchemaForGetColumns, 434
 - SupportsSchemaForGetTables, 434
 - SystemSchemas, 431
 - SystemTables, 431
 - TableTypes, 434
 - UseGetBestRowIdentifierForPrimaryKeys, 434
 - with schema tool, 447
 - Schema, 414
 - SchemaFactory, 414, 443
 - Schemas, 415
 - schemas list, 443
 - schema tool, 444
 - UseSchemaName, 434
 - XML representation, 447
- SchemaFactory, 414
- Schemas, 443
- schema tool
 - Ant task, 611
- Sequence, 177, 318, 411
 - allocate, 319
 - contiguous, 263
 - current, 318
 - Kodo extensions, 555
 - (see also generators)
 - metadata, 263
 - datastore-sequence attribute, 263
 - factory-class attribute, 264
 - name attribute, 263
 - strategy attribute, 263
 - next, 318
 - NextSequenceQuery, 435
 - nontransactional, 263
 - obtaining, 224

- runtime access, 558
 - Seq interface, 555
 - transactional, 263
 - SequenceGenerator, 126
 - allocationSize property, 126
 - initialValue property, 126
 - name property, 126
 - sequenceName property, 126
 - serialization, 12, 13, 173, 174
 - of enhanced types, 455
 - servlet, 583
 - ShortIdentity, 192
 - (see also single field identity)
 - SingleFieldIdentity, 191
 - (see also identity)
 - ByteIdentity, 191
 - CharIdentity, 192
 - IntIdentity, 192
 - LongIdentity, 192
 - ShortIdentity, 192
 - StringIdentity, 192
 - single field identity, 187, 190, 221
 - (see also identity)
 - single-string JDOQL (see JDOQL, single-string)
 - SQL, 15, 176, 213, 259
 - (see also JDOR)
 - (see also supported options set)
 - CatalogSeparator, 430
 - DDL (see DDL)
 - DistinctCountColumnSeparator, 431
 - executing with SQLLine, 449
 - ForUpdateClause, 431
 - InitializationSQL, 431
 - join syntax, 436
 - log messages, 417
 - queries (see SQL queries)
 - RequiresAliasForSubselect, 432
 - ReservedWords, 431
 - StringLengthFunction, 431
 - SubstringFunctionName, 431
 - SupportsSubselect, 432
 - TableForUpdateClause, 431
 - ToLowerCaseFunction, 431
 - ToUpperCaseFunction, 431
 - ValidationSQL, 431
 - SQL queries, 116, 320
 - (see also Query)
 - creating, 116, 320
 - named (see named queries)
 - parameters, 117, 322
 - projections, 322
 - result class, 323
 - retrieving persistent objects, 116, 321
 - stored procedures, 116, 321
 - SQLServer, 429
 - SQRT function, 88
 - StoreCache, 541
 - StoreCallback (see InstanceCallbacks)
 - stored procedures
 - as queries, 116, 321
 - (see also Query)
 - StoreLifecycleListener, 186
 - (see also lifecycle callbacks)
 - StringIdentity, 192
 - (see also single field identity)
 - Structured Query Language (see SQL)
 - SubclassFetchMode, 415, 474
 - SUBSTRING function, 87
 - sum, 244
 - (see also aggregates)
 - SunONE, 531
 - supported options set, 211
 - (see also PersistenceManagerFactory)
 - ApplicationIdentity, 212
 - Array, 212
 - ArrayList, 212
 - BinaryCompatibility, 213
 - ChangeApplicationIdentity, 212
 - DatstoreIdentity, 212
 - GetDataStoreConnection, 212
 - JDOQL, 213
 - LinkedList, 212
 - List, 212
 - NonDurableIdentity, 212
 - NontransactionalRead, 212
 - NontransactionalWrite, 212, 212
 - NullCollection, 212
 - Optimistic, 212
 - SQL, 213
 - TransientTransactional, 212
 - TreeMap, 212
 - TreeSet, 212
 - UnconstrainedQueryVariables, 213
 - Vector, 212
 - Sybase, 429
 - Synchronization, 228
 - (see also Transaction)
 - (see also transactions)
 - SynchronizeMappings, 415
- ## T
- TableGenerator, 127
 - allocationSize property, 127
 - catalog property, 127
 - initialValue property, 127
 - name property, 127
 - pkColumnName property, 127
 - pkColumnValue property, 127
 - schema property, 127
 - table property, 127
 - valueColumnName property, 127
 - TCP provider, 588
 - threading, 209, 216
 - (see also Multithreaded)
 - Multithreaded, 407
 - Transaction, 80, 176, 227, 227
 - (see also properties)
 - (see also transactions)

begin, 80, 228
commit, 80, 228
defaults, 209
demarcation, 80, 228
isActive, 81, 229
NontransactionalRead, 210, 228
NontransactionalWrite, 210, 228
obtaining from EntityManager, 70
obtaining from PersistenceManager, 216
Optimistic, 228
RestoreValues, 210, 228
RetainValues, 210, 228
rollback, 80, 228
RollbackOnly, 228
Synchronization, 228
transactional
 making nontransactional, 217
 making transactional, 217
 testing objects for, 202
 (see also JDOHelper)
TransactionIsolation, 415, 436
TransactionManager
 integration, 535
TransactionMode, 411, 535
transactions, 79, 202, 226
 (see also Transaction)
 ACID, 79, 226
 atomicity, 79, 226
 AutoClear
 AutoClear, 397
 checking consistency before commit, 223
 consistency, 79, 226
 datastore, 80, 210, 224, 227
 demarcating, 80, 228
 durability, 79, 226
 events, 559
 flushing changes before commit, 76, 222
 isolation, 79, 226, 436
 managed, 535
 ManagedRuntime, 406
 marking for rollback, 228
 optimistic, 72, 80, 201, 210, 210, 218, 227, 263, 408
 pessimistic, 79, 226
 synchronization callbacks, 228
 TransactionIsolation, 415
 TransactionMode, 411
 types, 79, 226
 XA, 536
Transient, 34
transient, 202, 202
 (see also lifecycle states)
 making transient, 217
transient-clean, 203
 (see also lifecycle states)
transient-dirty, 203
 (see also lifecycle states)
TransientTransactional, 212
 (see also supported options set)
transient-transactional, 203

 (see also lifecycle states)
transparent persistence, 173
TreeMap, 212
 (see also persistent fields)
 (see also supported options set)
TreeSet, 212
 (see also persistent fields)
 (see also supported options set)
trigger, 266, 266
TRIM function, 87

U

unbound variables (see variables)
unconstrained variables (see variables)
UnconstrainedQueryVariables, 213
 (see also supported options set)
 (see also variables)
unique constraints, 122, 312, 518
 deferred, 312
 SupportsUniqueConstraints, 430
uniqueness requirement, 22, 186
 (see also identity)
UpdateManager, 416
UPPER function, 87
user-defined
 persistent field types, 20, 182
 (see also persistent fields)
user objects
 PersistenceManager, 215
 (see also PersistenceManager)
uuid-hex, 35, 266
uuid-string, 35, 266

V

value object, 72, 219
variables, 236, 238
 (see also JDOQL)
 as query results, 243
 bound (see variables, constrained)
 constrained, 238
 implicit, 236
 unbound (see variables, unconstrained)
 unconstrained, 238
 checking for query support, 213
Vector, 212
 (see also persistent fields)
 (see also supported options set)
VendorName, 212
Version, 35
version, 284
 (see also locking)
 date-time strategy, 285
 mapping, 284, 508
 none strategy, 285
 retrieving from a persistent object, 201
 state-comparison strategy, 285
 surrogate, 482
 version-number strategy, 285

VersionColumn, 509
 (see also mapping metadata)
version fields, 19
 (see also persistent fields)
VersionNumber, 212
vertical, 131, 275
 (see also inheritance)

W

Weblogic, 527, 528, 529, 530
Websphere, 531
WriteLockLevel, 411, 542

X

XA transactions (see transactions)
XML, 193, 260