Oracle9*i*AS TopLink

Foundation Library Guide

Release 2 (9.0.3)

August 2002

Part No. B10064-01

ORACLE®

Oracle9*i*AS TopLink Foundation Library Guide Release 2 (9.0.3)

Part No. B10064-01

Contents

Send Us Your Comments xix

Preface xxi

1 Working with Database Sessions

Understanding Database sessions 1-2
- DatabaseSession class 1-2
- Creating a database session 1-3
  - Registering TopLink Mapping Workbench descriptors with a session 1-3
  - Registering Java descriptors with a session 1-3
  - Registering descriptors after login 1-3
  - Connecting to the database 1-4
  - Database interaction 1-4
  - Caching objects 1-4
  - Logging out of the database 1-4
- Logging SQL and messages 1-5
- Profiler 1-5
- Integrity checker 1-6
- Exception handlers 1-6
- JTS and external transaction controllers 1-7
- Creating database sessions: examples 1-7
- Reference 1-8

Using the Conversion Manager 1-9
Creating custom types with the Conversion Manager 1-10
Assigning custom classes to a TopLink session 1-10
The Conversion Manager class loader 1-10
Resolving class loader exceptions 1-11
Database login information 1-11
Creating a login object 1-11
Specifying database and driver information 1-12
Using the Sun JDBC-ODBC bridge 1-12
Using a different driver 1-13
Setting login parameters 1-13
Table Creator/Qualifier 1-14
Native SQL 1-14
Sequence number parameters 1-14
Binding and parameterized SQL 1-16
Batch writing 1-17
Data optimization 1-17
Cache isolation 1-18
Manual transactions 1-18
External transactions and connection pooling 1-18
Other database connections 1-18
Direct connect drivers 1-19
Using JDBC 2.0 data sources 1-19
Custom database connections 1-19
Building database logins: examples 1-20
Reference 1-21
Using the query framework 1-22
Session queries 1-22
Query objects 1-23
Custom SQL queries 1-24
Database exceptions 1-24
Querying on an inheritance hierarchy 1-25
Querying on interfaces 1-25

Using session queries ... 1-25
Reading objects from the database ... 1-26
Read operation ... 1-26
Read all operation ... 1-26
Refresh operation ... 1-27
Using expression builder ... 1-27
Using query by example ... 1-40
Writing objects to the database ... 1-40
Writing a single object to the database ... 1-40
Writing all objects to the database ... 1-41
Adding new objects to the database ... 1-41
Modifying existing objects in the database ... 1-41
Deleting objects in the database ... 1-41
Writing objects: Examples ... 1-42
Using transactions ... 1-42
Transaction operations ... 1-43
Nesting transactions ... 1-43
Implementing a transaction in Java code ... 1-43
Using units of work ... 1-44
Understanding the unit of work ... 1-44
Creating a unit of work ... 1-46
Registering existing objects with a unit of work ... 1-46
Reading objects using a unit of work ... 1-47
Creating new objects in a unit of work ... 1-47
Writing objects using a unit of work ... 1-48
Deleting objects through a unit of work ... 1-48
Resuming a unit of work ... 1-49
Reverting a unit of work ... 1-49
Executing queries from the unit of work ... 1-49
Nested and parallel units of work ... 1-50
Inside a unit of work ... 1-50
Advanced features ... 1-51
Read-only classes ... 1-51
Read-Only descriptors ... 1-52

Always Conform Descriptors 1-52
Merging 1-52
Validation 1-53
Troubleshooting the unit of work 1-54
Examples of units of work 1-54
Reference 1-55
Working with locking policies 1-56
Using optimistic lock 1-57
Advantages and disadvantages 1-57
Version locking policies 1-58
Field locking policies 1-58
Java implementation of optimistic locking 1-59
Advanced optimistic locking policies 1-60
Using optimistic read lock 1-60
Working with version fields 1-61
Pessimistic locking 1-64
Advantages and disadvantages 1-64
Reference 1-68
Session event manager 1-68
Session events 1-68
Using the session event manager: examples 1-70
Reference 1-70
Query objects 1-71
Query object components 1-72
Query types 1-72
Creating query objects 1-73
Executing queries 1-73
Query timeout 1-74
Read query objects 1-74
Parameterized SQL 1-75
Ordering for read all queries 1-75
Specifying the collection class 1-76
Using cursoring for a ReadAllQuery 1-76

Query optimization ... 1-76
Query return maximum rows ... 1-76
Partial object reading ... 1-77
Refreshing the identity map cache during a read query ... 1-77
In-memory querying and unit of work conforming ... 1-78
Conforming results in a unit of work ... 1-79
Handling exceptions resulting from in-memory queries ... 1-80
Disabling the identity map cache update during a read query ... 1-80
Internal query object caches ... 1-81
Write query objects ... 1-82
Non-cascading write queries ... 1-82
Disabling the identity map cache during a write query ... 1-83
Using query objects to customize the default database operations ... 1-83
Creating custom query operations ... 1-84
Using Query Redirectors ... 1-84
Reference ... 1-84
Query by example ... 1-85
Defining a sample instance ... 1-86
Defining a query by example policy ... 1-87
Combining query by example with expressions ... 1-88
Reference ... 1-88
Report query ... 1-89
Reference ... 1-90
Cursored streams and scrollable cursors ... 1-92
Java streams ... 1-92
Supporting streams ... 1-93
Using cursored streams and scrollable cursors: examples ... 1-93
Optimizing streams ... 1-94
Java iterators ... 1-94
Supporting scrollable cursor ... 1-95
Traversing scrollable cursors ... 1-95
SQL and stored procedure call queries ... 1-96
SQL Queries ... 1-96
Data-level queries ... 1-97
Stored procedure calls ... 1-97

Output parameters 1-98
Cursor output parameters 1-98
Output parameter event 1-98
Reference 1-99

2 Developing Enterprise Applications

Three-tier and enterprise applications 2-1
Client and server sessions 2-3
Client sessions 2-5
Server sessions 2-6
Caching database information on the server 2-6
Providing client read access 2-7
Providing client write access 2-8
Concurrency 2-9
Connection pooling 2-10
ServerSession connection options 2-11
Connection options 2-11
ClientSession connection options 2-11
Connection policies 2-12
Reference 2-12
Remote sessions 2-13
Architectural overview 2-15
Application layer 2-16
Transport layer 2-16
Server layer 2-17
Accessibility issues 2-17
Queries 2-18
Refreshing 2-18
Indirection 2-18
Cursored streams 2-19
Unit of work 2-19
Creating a remote connection using RMIConnection 2-19
Session broker 2-20
Two-phase/two-stage commits 2-21
Using the session broker 2-21

Using the session broker in a three-tier architecture ... 2-22
Creating multiple projects in the Mapping Workbench ... 2-23
Limitations ... 2-24
Advanced use ... 2-24
Reference ... 2-24
Java Transaction Service (JTS) ... 2-25
Review of transactions and transaction management ... 2-25
Distributed transactions ... 2-26
Transaction managers ... 2-26
Two-phase commit with presumed rollback ... 2-27
Relationship between OMG Object Transaction Service (OTS) and Java Transaction Service (JTS) 2-28
JTS transaction synchronization ... 2-29
TopLink unit of work and the synchronization interface ... 2-29
Writing to a database in three-tier environment ... 2-30
External connection pools and external transaction control ... 2-31
Extending TopLink's JTS capabilities ... 2-33
TopLink support for Java Data Objects (JDO) ... 2-37
Understanding the JDO API ... 2-37
JDO implementation ... 2-38
JDOPersistenceManagerFactory ... 2-38
JDOPersistenceManager ... 2-41
JDOQuery ... 2-45
JDOTransaction ... 2-51
Running the TopLink JDO demo ... 2-53
Distributed Cache Synchronization ... 2-53
Controlling the sessions: the Cache Synchronization Manager ... 2-54
Using Cache Synchronization Manager options ... 2-54
Using a clustering service ... 2-55
Configuring cache synchronization ... 2-56
Connecting the sessions ... 2-57
Using Java Messaging Service ... 2-58
Preparing to use JMS ... 2-58
Setting up JMS in the session configuration file ... 2-59
Setting up JMS in Java ... 2-59

3 Working with Enterprise JavaBeans

The EJB specification 3-1
Additional information 3-2
Using the session bean model 3-2
Session beans and DatabaseSessions 3-4
Interactions with JTS 3-4
Using session beans with TopLink's three-tier application model 3-5
Using the Session Manager 3-5
Retrieving a session from a SessionManager 3-6
Using the default configuration file: sessions.xml 3-6
Using the XMLLoader 3-7
Using the entity bean model 3-9
TopLink and container-managed persistent entity beans 3-10

4 EJBQL Support

Why use EJBQL? 4-1
EJBQL structure 4-2
Basic structure 4-2
The FROM clause 4-2
The FROM clause defined 4-3
Using the FROM clause: a few examples 4-3
The SELECT clause 4-4
Using the SELECT clause: a few examples 4-4
The WHERE clause 4-5
Using constants 4-6
Comparison Operators 4-6
Logical operators 4-7
Null Comparison Expressions: Null 4-9
Range Expressions 4-9
Functional Expressions 4-10
Input Parameters 4-11
Combining Clauses 4-12
Multiple clauses: a few examples 4-12

Using EJBQL with TopLink 4-12
ReadAllQuery 4-13
Session 4-13

5 SDK for XML and Non-relational Database Access

Using the TopLink SDK 5-1
Accessor 5-2
Data Store Connection 5-2
Call Execution 5-3
Transaction Processing 5-3
Calls 5-3
Read Object Call 5-4
Read All Call 5-5
Insert Call 5-5
Update Call 5-5
Delete Call 5-5
Does Exist Call 5-5
Custom Call 5-5
Database Row 5-6
FieldTranslator 5-7
SDKDataStoreException 5-8
Descriptors and Mappings 5-8
SDKDescriptor 5-8
Standard mappings 5-11
SDK Mappings 5-15
Sessions 5-25
SDKPlatform 5-25
SDKLogin 5-26
TopLink Project 5-27
Session 5-27
Unsupported features 5-28
Using TopLink XML support 5-28
Getting Started 5-28
Configure your Login using an XMLFileLogin. 5-28
Build your Project. 5-29

Build your Descriptors using XMLDescriptors 5-29
Build your Mappings 5-29
Build your DatabaseSession and log in. 5-30
Build your sequences, if necessary 5-30
Use the Session 5-30
Customizations 5-31
Implementation details 5-31
XMLFileAccessor 5-32
XMLAccessor implementation 5-32
Directory creation 5-33
XMLCall 5-33
XMLStreamPolicy 5-33
XMLTranslator 5-34
XMLTranslator implementations 5-34
XMLDescriptor 5-37
XMLPlatform 5-37
XMLFileLogin 5-38
XMLSchemaManager 5-38
XMLAccessor 5-38
XMLTranslator 5-39
DefaultXMLTranslator 5-39
SDKAggregateObjectMapping 5-40
SDKDirectCollectionMapping 5-40
XML Zip File Extension 5-42
Using the Zip file extension 5-42
Configure direct file access with Zip File extension 5-43
Implementation details 5-43

6 Performance Optimization

Basic performance optimization 6-1
TopLink reading optimization features 6-2
Reading Case 1: Displaying names in a list - optimized through partial object reading and report query 6-3
Partial object reading 6-3
Conclusion 6-5

Reading Case 2: Batch reading objects 6-6
Conclusion 6-8
Reading Case 3: Using complex custom SQL queries 6-8
Reading Case 4: Viewing objects 6-8
TopLink writing optimization features 6-10
Writing Case 1: Batch writes 6-11
Batching and cursoring 6-12
Sequence number pre-allocation 6-12
Batch writing 6-13
Parameterized SQL 6-13
Multi-processing 6-13
Optimization check list 6-15
Schema optimization 6-15
Schema Case 1: Aggregation of two tables into one 6-16
Domain 6-16
Schema Case 2: Splitting one table into many 6-17
Domain 6-18
Schema Case 3: Collapsed hierarchy 6-18
Domain 6-19
Schema Case 4: Choosing one out of many 6-20
Domain 6-20

7 Mapping Implementation

Direct mappings 7-1
Direct-to-field mappings 7-2
Reference 7-3
Type conversion mappings 7-3
Reference 7-4
Object type mappings 7-4
Reference 7-5
Serialized object mappings 7-6
Reference 7-6
Transformation mappings 7-7
Implementing transformation mappings in Java 7-7
Reference 7-10

Relationship mappings 7-10
Aggregate object mappings 7-11
Reference 7-12
One-to-one mappings 7-13
Reference 7-14
Variable one-to-one mappings 7-15
Reference 7-16
Direct collection mappings 7-16
Reference 7-17
Aggregate collections 7-18
When to use aggregate collections 7-18
Aggregate collections and inheritance 7-19
Java implementation 7-19
Reference 7-20
Direct map mappings 7-21
Reference 7-22
One-to-many mappings 7-23
Reference 7-24
Many-to-many mappings 7-24
Reference 7-25
Object relational mappings 7-26
Array mappings 7-26
Implementing array mappings in Java 7-27
Reference 7-27
Object array mappings 7-28
Implementing object array mappings in Java 7-28
Reference 7-29
Structure mappings 7-29
Implementing structure mappings in Java 7-30
Reference 7-31
Reference mappings 7-31
Implementing reference mappings in Java 7-32
Reference 7-32
Nested table mappings 7-33
Implementing nested table mappings in Java 7-33
Reference 7-34

8 Descriptor Implementation

Implementing primary keys in Java 8-1
Implementing inheritance in Java 8-2
Reference 8-8
Implementing interfaces in Java 8-8
Setting the copy policy using Java 8-9
Implementing multiple tables in Java code 8-9
Primary keys match 8-9
Primary keys are named differently 8-10
Tables related by foreign key relationships 8-11
Non-standard table relationships 8-12
Implementing sequence numbers in Java 8-14
Overriding the instantiation policy using Java code 8-15
Implementing locking in Java 8-16
Implementing identity maps in Java 8-17
Implementing query keys in Java 8-17
Implementing indirection in Java 8-18
Implementing proxy indirection in Java 8-19
Implementing object-relational descriptors in Java 8-19
Changing Java classes to use indirection 8-20
Setting the wrapper policy using Java code 8-21
Implementing events using Java 8-21
Registering event listeners 8-22
Reference 8-22

A Sessions.xml DTD

sessions.xml dtd A-1

B TopLink Development Tools

The Schema Manager B-1
Using the Schema Manager to create tables B-2
Creating a table definition B-2
Adding fields to a table definition B-2
Defining Sybase and SQL Server sequence numbers B-3

Example of table definition B-4
Creating tables on the database B-4
Creating the sequence table B-5
Using the Schema Manager to manage Java and database type conversions B-5
Session management services B-9
RuntimeServices B-9
DevelopmentServices B-9
Using session management services B-9
The stored procedure generator B-10
Generation of stored procedures B-10
Attaching the stored procedures to the descriptors B-11
The Session Console B-11
Requirements B-12
Using session console features B-12
Launching the session console from code B-15
The Performance Profiler B-15
Using the profiler B-16
Browsing the profiler results B-17

C TopLink Session Configuration File

Contents of the sessions.xml file C-1
Converting from TOPLink.properties file to sessions.xml C-3

D EJBQL Syntax

About Backus Naur Form D-1
EJBQL language definition D-2

Index

Send Us Your Comments

Oracle9*i*AS TopLink Foundation Library Guide Release 2 (9.0.3)

Part No. B10064-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: iasdocs_us@oracle.com
- FAX: 650-506-7407 Attn: Oracle9*i* Application Server Documentation Manager
- Postal service:
  Oracle Corporation
  Oracle9*i* Application Server Documentation
  500 Oracle Parkway, M/S 2op3
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This section introduces the information you need to get the most out of the documentation that accompanies your software. This preface contains these topics:

- Intended Audience
- Documentation Accessibility
- Structure
- Related Documents
- Conventions

Intended Audience

This document is intended for application developers who perform the following tasks:

- Application design and development
- Application testing and benchmarking
- Application integration

This document assumes that you are familiar with the concepts of object-oriented programming, the Enterprise JavaBeans (EJB) specification, and with your own particular Java development environment.

The document also assumes that you are familiar with your particular operating system (Windows, UNIX, or other). The general operation of any operating system is described in the user documentation for that system, and is not repeated in this manual.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Structure

This document contains:

The following shows a sample presentation of structure:

Chapter 1, "Working with Database Sessions"

This chapter is a comprehensive reference for database sessions in TopLink. It describes the fundamental concepts required to connect to the database and to perform queries as well as optional and advanced session and query properties.

Chapter 2, "Developing Enterprise Applications"

This chapter describes how to develop enterprise applications using TopLink, and discusses the issues and techniques associated with creating Enterprise applications. It also illustrates some of the TopLink features that enable TopLink to integrate with industry-leading enterprise application servers, including Oracle9*i*AS.

Chapter 3, "Working with Enterprise JavaBeans"

This chapter describes TopLink features that provide support for Enterprise JavaBeans. It discusses a number of topics related to TopLink support for EJBs and EJBs in general.

Chapter 4, "EJBQL Support"

This chapter describes TopLink's support for EJBQL, and includes a discussion on the advantages and disadvantages of using EJQBQL in s TopLink query.

Chapter 5, "SDK for XML and Non-relational Database Access"

This chapter describes the TopLink Software Development Kit (SDK), which provides support for non-relational database access and eXtensible Markup Language (XML).

Chapter 6, "Performance Optimization"

This chapter discusses how to optimize TopLink-enabled applications for best performance. It includes sections on basic performance optimization, writing optimization, and schema optimization.

Chapter 7, "Mapping Implementation"

This chapter describes how to implement mappings in Java code for TopLink-based applications.

Chapter 8, "Descriptor Implementation"

This chapter describes how to implement descriptors in Java code for TopLink-based applications.

Appendix A, "Sessions.xml DTD"

This appendix contains the DTD for `sessions.xml`, the session configuration file used by TopLink sessions.

Appendix B, "TopLink Development Tools"

This appendix contains information on the development tools that make the development, testing and debugging of TopLink applications easier.

Appendix C, "TopLink Session Configuration File"

This appendix contains a description of all elements available in the TopLink `sessions.xml` file.

Appendix D, "EJBQL Syntax"

This appendix describes the TopLink implementation of EQJBQL, and includes a discussion of EJBQL syntax.

Related Documents

For more information, see these Oracle resources:

Oracle9*i*AS TopLink Getting Started

Provides installation procedures to install and configure TopLink. It also introduces the concepts with which you should be familiar to get the most out of TopLink.

Oracle9*i*AS TopLink Tutorials

Provides tutorials illustrating the use of TopLink. It is written for developers who are familiar with the object-oriented programming and Java development environments.

Oracle9*i*AS TopLink Foundation Library Guide

Introduces TopLink and the concepts and techniques required to build an effective TopLink application. It also gives a brief overview of relational databases and describes who TopLink accesses relational databases from the object-oriented Java domain.

Oracle9*i*AS TopLink Mapping Workbench Reference Guide

Includes the concepts required for using the TopLink Mapping Workbench, a stand-alone application that creates and manages your descriptors and mappings for a project. This document includes information on each Mapping Workbench function and option and is written for developers who are familiar with the object-oriented programming and Java development environments.

Oracle9*i*AS TopLink Container Managed Persistence for Application Servers

Provides information on TopLink container-managed persistence (CMP) support for application servers. Oracle provides an individual document for each application server specifically supported by TopLink CMP.

Oracle9*i*AS TopLink Troubleshooting

Contains general information about TopLink's error handling strategy, the types of errors that can occur, and Frequently Asked Questions (FAQs). It also discusses troubleshooting procedures and provides a list of the exceptions that can occur, the most probable cause of the error condition, and the recommended action.

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://otn.oracle.com/admin/account/membership.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://otn.oracle.com/docs/index.htm`

Conventions

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.

Convention	Meaning	Example
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id` and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example
. . .	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , coln FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

Conventions for Microsoft Windows Operating Systems

The following table describes conventions for Microsoft Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose **Start** >	How to start a program.	To start the Oracle Database Configuration Assistant, choose **Start** > **Programs** > **...** .
Case sensitivity and file and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \, then Windows assumes it uses the Universal Naming Convention.	`c:\winnt"\"system32` is the same as `C:\WINNT\SYSTEM32`
	IMPORTANT NOTE: File names and directory names *are* case sensitive under UNIX. Where the name of a file or directory is mentioned and the operating system is a non-Windows platform, you must enter the names exactly as they appear unless instructed otherwise.	

Convention	Meaning	Example
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`

	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\>exp scott/tiger TABLES=emp QUERY=\"WHERE job='SALESMAN' and sal<1600\"`
		`C:\>imp SYSTEM/`*`password`*
		`FROMUSER=scott TABLES=(emp, dept)`

`<`*`INSTALL_DIR`*`>`	Represents the Oracle home installation directory name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`SET CLASSPATH=<`*`INSTALL_DIR`*`>\jre\bin`

Convention	Meaning	Example
ORACLE_HOME and *ORACLE_BASE*	In releases prior to Oracle8*i* release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *ORACLE_HOME* directory that by default used one of the following names: ▪ `C:\orant` for Windows NT ▪ `C:\orawin95` for Windows 95 ▪ `C:\orawin98` for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *ORACLE_HOME* directory. There is a top level directory called *ORACLE_BASE* that by default is `C:\oracle`. If you install Oracle9*i* release 1 (9.0.1) on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is `C:\oracle\ora90`. The Oracle home directory is located directly under *ORACLE_BASE*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle9i Database Getting Starting for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *ORACLE_BASE*`\`*ORACLE_HOME*`\rdbms\admin` directory.

1

Working with Database Sessions

A database session represents an application's dialog with a relational database. This chapter is a comprehensive reference for database sessions in TopLink. It describes the fundamental concepts required to connect to the database and to perform queries as well as optional and advanced session and query properties. It discusses

- Understanding Database sessions
- Database login information
- Using the query framework
- Using session queries
- Writing objects to the database
- Using units of work
- Working with locking policies
- Session event manager
- Query objects
- Query by example
- Report query
- Cursored streams and scrollable cursors
- SQL and stored procedure call queries

You should have a good command of the topics in this chapter and the Descriptors and Mappings chapters before using TopLink in an application.

A complete listing of the TopLink application programming interface (API) is provided in HTML format. It is located in the Java Docs directory where TopLink was installed. Refer to this document for more information on the complete TopLink API.

Understanding Database sessions

A *session* represents the connection between an application and the relational database that stores its persistent objects. TopLink provides several different session objects that all implement the same Session interface. The simplest session is the `DatabaseSession`, which can be used for single user/single database applications. All of the following examples use `DatabaseSession`.

TopLink also provides a `ServerSession`, `ClientSession`, `RemoteSession`, `UnitOfWork` and `SessionBroker`. For more information on these sessions, refer to Chapter 2, "Developing Enterprise Applications".

Note: If you are building a three-tier application, use the `ServerSession`, not a `DatabaseSession`. If you use `DatabaseSession`, it may be difficult to migrate your application to a scalable architecture in the future.

DatabaseSession class

An application must create an instance of the `DatabaseSession` class. A `DatabaseSession` class stores the following information:

- an instance of `Project` and `DatabaseLogin`, which stores database login and configuration information
- an instance of `DatabaseAccessor`, which wraps the JDBC connection and handles database access
- the descriptors for each of the application's persistent classes
- the identity maps, which maintain object identity and act as a cache

The session is created from an instance of `Project`, which contains the database connection parameters.

- If the project was generated or read from the TopLink Mapping Workbench project, its descriptors are automatically loaded into the session.
- If the descriptors were built using code, or if multiple projects were used, the application must register the descriptors of the persistent classes with the session before logging in to the database.

A typical application then reads from the database using the TopLink query framework, and writes to the database using a unit of work. A well-designed application then logs out of the database when it is finished accessing the persistent objects in the database.

Creating a database session

Instances of `DatabaseSession` must be created from a `Project` instance. Initialize this project with all of the appropriate database login parameters, such as the JDBC driver and the database URL. Refer to "Understanding Database sessions" on page 1-2 for more information on reading the TopLink Mapping Workbench project file.

Registering TopLink Mapping Workbench descriptors with a session

If the application uses descriptors created with the TopLink Mapping Workbench tool, the project adds its descriptors automatically. If multiple projects are used, the additional projects must use the `addDescriptors(Project)` method to register their descriptors with the session. Refer to "Understanding Database sessions" on page 1-2 for more information on reading the TopLink Mapping Workbench project file.

Registering Java descriptors with a session

If the application does not use a TopLink Mapping Workbench project, register a `Vector` of descriptors using the `addDescriptors(Vector)` method.

> **Note:** You can also register each descriptor individually using the addDescriptor(Descriptor) method; however, registering descriptors using a vector minimizes the possibility of errors.

Registering descriptors after login

Descriptors can be registered after the session logs in, but they should be independent of any descriptors already registered. This allows self-contained sub-systems to be loaded after connecting.

It is also possible to re-register descriptors that have already been loaded. If this is done, ensure that all related descriptors are re-registered at the same time. Changes to one descriptor may affect the initialization of other descriptors.

Connecting to the database

After the descriptors have been registered, the `DatabaseSession` can connect to the database using the `login()` method. If the login parameters in the `DatabaseLogin` class are incorrect, or if the connection cannot be established, a `DatabaseException` is thrown.

After a connection is established, the application is free to use the session to access the database. The `isConnected()` method returns true if the session is connected to the database.

Database interaction

The application can interact with the database using the session's querying methods or by executing query objects. The interactions between the application and the database are collectively called the query framework. Refer to "Using the query framework" on page 1-22 for more information on querying.

Caching objects

Database sessions have an identity map, which maintains object identity and acts as a cache. When an object is read from the database it is instantiated and stored in the identity map. If the application queries for the same object, TopLink returns the object in the cache rather than reading the object from the database again.

The `initializeIdentityMaps()` method can be called to flush all objects from the cache.

> **Caution:** When using this method, make sure that none of the objects in the cache are in use.

The identity map can be customized for performance reasons. Refer to the *Oracle9iAS TopLink Mapping Workbench Reference Guide* for more information on using the identity map and caching.

Logging out of the database

The session can log out using the `logout()` method. Since logging in to the database can be time consuming, log out only when all database interactions are complete.

When the `logout()` method is called, the session is disconnected from the relational database, and its identity maps are flushed. Applications that log out do not have to register the descriptors again when they log back in to the database.

Logging SQL and messages

TopLink accesses the database by generating SQL strings. TopLink handles all SQL generation internally, and applications that use the session methods or query objects do not have to deal with SQL. For debugging purposes, programmers who are familiar with SQL may wish to keep a record of the SQL used to access the database.

The `DatabaseSession` class provides methods to allow the SQL generated to be logged to a writer. SQL and message logging is disabled by default, but can be enabled using the `logMessages()` method on the session. The default writer is a stream writer to `System.out`, but the writer can be changed using the `setLog()` method of the session.

The session can log:

- the state of the cache
- the state of a unit of work. This can be done using the `printIdentityMaps()` and `printRegisteredObjects()` methods.
- debug print statements
- exceptions/error messages sent to system out

and any other output sent to the system log.

Profiler

TopLink offers a high-level logging service called the Profiler. Instead of logging raw SQL statements, the profiler can be enabled to log a summary of each query that is executed. This summary includes a performance breakdown of the query to easily identify performance bottlenecks and has been extended to provide more granularity with regards to the query information provided. A report that summarizes the querying performance for an entire session can also be logged from the profiler.

TopLink also provides a GUI browser for profiles that can be accessed through the session console.

Refer to Appendix B, "TopLink Development Tools" for more information on the profiler and session console.

Integrity checker

When a session is connected or descriptors are added to a session after it is connected, TopLink initializes and validates the descriptor's information. The integrity checker allows for the validation process to be customized.

By default, the integrity checker reports all errors discovered with the descriptors during initialization. The integrity checker can be configured to:

- throw the first error that it encounters, including the error's stack trace
- validate the state of the database schema to ensure it matches the information in the descriptors
- disable the instance creation check

Example 1–1 Using the integrity checker

```
session.getIntegrityChecker().checkDatabase();
session.getIntegrityChecker().catchExceptions();
session.getIntegrityChecker().dontCheckInstantiationPolicy();
session.login();
```

Exception handlers

Exception handlers can be used on the session to handle database exceptions. An implementor of the `ExceptionHandler` interface can be registered with the session. When a database exception occurs during the execution of a query, the exception is passed to the exception handler instead of being thrown. The exception handler can then decide to handle the exception, retry the query, or throw an unchecked exception. Exception handlers are typically used to handle connection timeouts or database failures. See *Oracle9iAS TopLink Troubleshooting* for more information on exceptions.

Example 1–2 Implementing an exception handler

```
session.setExceptionHandler(newExceptionHandler(){
    public Object handleException(RuntimeException exception) {
        if ((exception instanceof DatabaseException) &&
            (exception.getMessage().equals("connection reset by peer."))) {
            DatabaseException dbex = (DatabaseException) exception;
            dbex.getAccessor().reestablishConnection (dbex.getSession());
            return dbex.getSession().executeQuery(dbex.getQuery());
        }
        return null;
    }
```

```
});
```

JTS and external transaction controllers

For detailed information on Java Transaction Service (JTS) and external transaction controllers, see Java Transaction Service (JTS) on page 2-28.

Creating database sessions: examples

Example 1–3 Creating and using a session from a TopLink Mapping Workbench project

```
import oracle.toplink.tools.workbench.*;
import oracle.toplink.sessions.*

// Create the project object.
Project project = ProjectXMLReader.read("C:\TopLink\example.xml");
DatabaseLogin loginInfo = project.getLogin();
loginInfo.setUserName("scott");
loginInfo.setPassword("tiger");

//Create a new instance of the session and login.
DatabaseSession session = project.createDatabaseSession();
try {
    session.login();
    } catch (DatabaseException exception) {
    throw new RuntimeException("Database error occurred at login: " +
    exception.getMessage());
    System.out.println("Login failed");
    }

// Do any database interaction using the query framework, transactions or units
of work.
...

// Log out when database interaction is over.
session.logout();
Creating and using a session from coded descriptors
import oracle.toplink.sessions.*;

//Create the project object.
DatabaseLogin loginInfo = new DatabaseLogin();
loginInfo.useJDBCODBCBridge();
loginInfo.useSQLServer();
```

```
loginInfo.setDataSourceName("MS SQL Server");
loginInfo.setUserName("scott");
loginInfo.setPassword("tiger");
Project project = new Project(loginInfo);

//Create a new instance of the session, register the descriptors, and login.
DatabaseSession session = project.createDatabaseSession();
session.addDescriptors(this.buildAllDescriptors());
try {
    session.login();
} catch (DatabaseException exception) {
    throw new RuntimeException("Database error occurred at login: " +
    exeption.getMessage());
    System.out.println("Login failed");
    }

//Do any database interaction using the query framework, transactions or units
of work.
...
//Log out when database interaction is over.
session.logout();
```

Reference

Table 1–1 summarizes the most common public methods for the `DatabaseSession` class:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for the `DatabaseSession` class, see the TopLink JavaDocs.

Table 1–1 Elements for DatabaseSession

Element	Default	Method Names
Construction methods	not applicable	`Project.createDatabaseSession()`
Log into the database	user name and password from project login	`login()`
Log out of the database	not applicable	`logout()`
Executing predefined queries		`executeQuery(String queryName)`

Table 1–1 Elements for DatabaseSession (Cont.)

Element	Default	Method Names
Executing a query object		`executeQuery(DatabaseQuery query)`
Reading from the database	not applicable	`readAllObjects(Class domainClass, Expression expression)` `readObject(Class domainClass, Expression expression)`
SQL logging	do not log SQL	`logMessages()`
Debugging		`printIdentityMaps()`
Identity maps (advanced)	not applicable	`getFromIdentityMap(Vector primaryKey, Class theClass)`

Transactions	not applicable	`beginTransaction()`
		`commitTransaction()`
		`rollbackTransaction()`

Exception handlers	throw exception	`setExceptionHandler(ExceptionHandler handler)`
JTS	JDBC transactions	`setExternalTransactionController (ExternalTransactionController controller)`
Unit of work	not applicable	`acquireUnitOfWork()`
Writing to the database	not applicable	`deleteObject(Object domainObject)` `writeObject(Object domainObject)`

Using the Conversion Manager

TopLink uses a class called the ConversionManager to convert database types to Java types. This class, found in the `oracle.toplink.internal.helper` package, is the central location for type conversion, and as such can provide the expert developer with a mechanism for using custom types within TopLink.

Creating custom types with the Conversion Manager

To use custom types, create a subclass of the ConversionManager. Do one of the following:

- Overload the `public Object convertObject(Object sourceObject, Class javaClass)` method to call the conversion method that is written in the subclass for the custom type.
- Delegate the conversion to the super class.

The conversion method, `protected ClassX convertObjectToClassX(Object sourceObject) throws ConversionException` must be implemented to convert incoming object to the required class.

Assigning custom classes to a TopLink session

Once the class has been written, assign it to TopLink. There are two common ways to accomplish this:

- A TopLink session can be assigned a custom Conversion Manager through the `(getSession().getPlatform().setConversionManager(ConversionManager))` platform.
- Set the Singleton to use the custom Conversion Manager by calling the static method on the Conversion Manager `setDefaultManager(ConversionManager)`. By setting this before any session are logged in, all TopLink sessions created in a particular VM will use the custom Conversion Manager. See the ConversionManager class JavaDocs for examples.

The Conversion Manager class loader

The Conversion Manager loads the classes included in a mapping project, as well as classes throughout the library. TopLink provides storage of a class loader within the Conversion Manager to facilitate this. The class loader in the Conversion Manager is set to the System class loader by default.

Resolving class loader exceptions

There are cases, particularly when TopLink is deployed within an application server, when other class loaders are used for the deployed classes. In these cases, a `ClassNotFound` exceptions may be thrown. To resolve this problem, do one of the following:

- Call `public void setShouldUseClassLoaderFromCurrentThread (boolean useCurrentThread)` on the default Conversion Manager before logging in any sessions. This resolves the problem for most application servers, and ensures that TopLink uses the correct ClassLoader.
- Set the default class loader to the class loader used by the ApplicationLoader; for example, if the Session Manager is being used, the class Loader can be passed into the `getSession()` call, which sets the required class loader on the Conversion Manager.
- Call `public static void setDefaultLoader (ClassLoader classLoader)` on the default Conversion Manager before any sessions are logged in, passing in the ClassLoader that contains the deployed classes.

Database login information

Java applications that access a database log in to the database through a JDBC driver. To login successfully, the database typically requires a valid username and password. In a TopLink application, this login information is stored in the `DatabaseLogin` class. All sessions must have a valid `DatabaseLogin` instance before logging in to the database.

This section describes the basic login properties and also the various advanced configuration options available on `DatabaseLogin`. The advanced options are normally not required unless the JDBC driver being used is not fully JDBC compliant.

Creating a login object

The Project class you create must include a login object to access the database used by the project. The most basic login mechanism involves creating an instance of `DatabaseLogin` through its default constructor, as follows:

```
Databaselogin login = new Databaselogin
...
```

Caution: This basic method of database login should only be used when the project was not created in the TopLink Mapping Workbench.

The `Project` class provides the `getLogin()` instance method to return the project's login. This method returns an instance of `DatabaseLogin`. The `DatabaseLogin` object can then be used directly or be provided with more information before logging in.

However, if you create the project in the TopLink Mapping Workbench, the login object is created automatically for you. In this case, you should only access the login from your Project instance. This ensures that login information set in TopLink Mapping Workbench, such as sequencing information, is used by the session, and also prevents you from inadvertently over-writing the login information already included in the project

Specifying database and driver information

The `DatabaseLogin` method assumes that the database being accessed is a generic JDBC-compliant database. TopLink also provides custom support for most database platforms. To take advantage of this support, you can call the `use`*`XDriver()`* method for your specific platform along with the `getLogin()` instance method:

```
project.getLogin().useOracle();
```

The `DatabaseLogin` class has several helper methods, such as `useJConnectDriver()`, that set the driver class, driver URL prefix, and database information for common drivers. If one of these helper methods is used, only the database instance-specific part of the JDBC driver URL needs to be specified, using the `setDatabaseURL()` method. These helper methods also set any additional settings required for that driver, such as binding byte arrays or using native SQL. They are recommended for specifying your driver information. For example:

```
project.getLogin().useOracleThinJDBCDriver();
project.getLogin().setDatabaseURL("dbserver:1521:orcl");
```

By default, new `DatabaseLogin` objects use the Sun JDBC-ODBC bridge. However, if you require a different driver, you can specify a different connection mechanism.

Using the Sun JDBC-ODBC bridge

If you are using the Sun JDBC-ODBC bridge, only the ODBC datasource name is required. Call `setDataSourceName()` to specify it. A list of your installed data sources can be found from the "ODBC Administrator" in your Windows control panel. For example:

```
project.getLogin().useJDBCODBCBridge();
    project.getLogin().useOracle();
    project.getLogin().setDataSourceName("Oracle");
```

Note: TopLink splits the URL into the driver portion and the database portion when using this method. The `setConnectionString()` function can also be used to set the entire URL.

Using a different driver

If you require a driver other than the Sun JDBC-ODBC bridge, you can specify a different connection mechanism by calling the `setDriverClass()` and `setConnectionString()` methods to indicate which driver to use.

For example:

```
project.getLogin().setDriverClass(oracle.jdbc.driver.OracleDriver.class);
project.getLogin().setConnectionString("jdbc:oracle:thin:@dbserver:1521:orcl");
```

Note: Refer to the documentation supplied with your driver to determine the correct settings to use with these methods.

Setting login parameters

If the database requires user and password information, the application must call `setUserName()` and `setPassword()`. This must be done after the driver has been specified. This is normally required when using the login from your TopLink Mapping Workbench project, as the Mapping Workbench does not store the password by default.

Example 1–4 Using setUserName() and setPassword()

```
project.getLogin().setUserName("userid");
project.getLogin().setPassword("password");
```

Properties such as the database name and the server name may be specified through the `setServerName()` and `setDatabaseName()` methods. Most JDBC drivers do *not* require the database and server name properties because they are part of the database URL. Specifying them can cause connection failures, so avoid setting them unless using JDBC-ODBC. Only some JDBC-ODBC bridges require these properties to be set. They are usually set from the ODBC Data Source Administrator, so they are normally not required.

Some JDBC drivers require additional properties that are not mentioned here. The additional properties can be specified through the `setProperty()` method. Also, some drivers fail to connect if properties are specified when not required. If a connection always fails, check to make sure the properties are correct.

Note: Do not set the login password directly using the `setProperty()` method, as TopLink encrypts and decrypts the password. Use the `setPassword()` method instead.

Table Creator/Qualifier

The `setTableQualifier()` method can be used to prepend a given string to all tables accessed by the session. This is useful for setting the name of the table creator, or owner, for databases such as Oracle and DB2. This should be used when a common user such as DBA defined the entire schema. If some tables have a different creator, the table name must be fully qualified with the creator in the descriptor.

Native SQL

By default, TopLink accesses the database using JDBC SQL. The JDBC syntax uses "{" escape clauses to print dates and binary data. If your driver does not support this syntax you will get an error on execution of SQL that contains dates.

To use native SQL for database interaction, call the `useNativeSQL()` method. This is required only if your JDBC driver does not support the JDBC standard SQL syntax, such as Sybase JConnect 2.x. Because native SQL is database-specific, ensure that you have set your database platform to the correct database.

Example 1–5 Using native SQL with a Sybase database

```
project.getLogin().useSybase();
project.getLogin().useNativeSQL();
```

Sequence number parameters

You can specify sequencing information in the `DatabaseLogin` by using the following methods:

- `setSequenceCounterFieldName()`
- `setSequenceNameFieldName()`
- `setSequencePreallocationSize()`

- `setSequenceTableName()`
- `useNativeSequencing()`

If your application uses native sequencing rather than a sequence table, call the `useNativeSequencing()` method. TopLink supports native sequencing on Oracle, Sybase, SQL Server and Informix. The database platform must have been specified to use native sequencing.

- When using native sequencing, the sequence pre-allocation size defaults to 1. If Sybase, SQL Server or Informix native sequencing is used, then pre-allocation cannot be used and the size must not be changed.
- When using native sequencing with Oracle, the name of the sequence object used to generate the sequence numbers must be configured in each descriptor using sequencing and the sequence pre-allocation size must match the "increment" on the sequence object.

Notes:

- Be sure to match the "increment" of the Oracle sequence and not the "cache". The cache refers to the sequences cached on the database server, while the increment refers to the number of sequences that can be cached on the database client.
- When using sequencing or native sequencing, the sequence information must also be specified in each descriptor that makes use of a generated id.
- It is recommended to always use pre-allocation and to only use native sequencing in Oracle since native sequencing in other databases does not support pre-allocation.

Example 1–6 Using native sequencing

```
project.getLogin().useOracle();
project.getLogin().useNativeSequencing();
project.getLogin().setSequencePreallocationSize(1);
```

Note: Using the `Project` class to create a `DatabaseLogin` instance automatically sets the sequencing information specified in TopLink Mapping Workbench.

Refer to the *Oracle9iAS TopLink Mapping Workbench Reference Guide* for more information on sequence numbers.

Binding and parameterized SQL

By default, TopLink prints data inlined into the SQL it generates and does not use parameterized SQL. The difference between parameter binding and printing data is that some drivers have limits on the size of data that can be printed. Also, parameterized SQL allows for the prepared statement to be cached to improve performance. Many JDBC drivers do not fully support parameter binding or have size or type limits. Refer to your database documentation for more information on binding and binding size limits.

TopLink can be configured to use parameter binding for large binary data with the `useByteArrayBinding()` method. Some JDBC drivers function better if large binary data is read through streams. For this purpose, TopLink can also be configured to use streams for binding by calling the `useStreamsForBinding()` method. Binding can also be configured for large string data through the `useStringBinding()` method.

TopLink supports full parameterized SQL and prepared statement caching, both of which are configured through the `bindAllParameters()`, `cacheAllStatements()` and `setStatementCacheSize()`methods. Refer to Chapter 6, "Performance Optimization" for more information on parameterized SQL.

Example 1–7 Using parameter binding with large binary data

```
project.getLogin().useByteArrayBinding();
    project.getLogin().useStreamsForBinding();
        project.getLogin().useStringBinding(50);
    project.getLogin().bindAllParameters();
        project.getLogin().cacheAllStatements();
        project.getLogin().setStatementCacheSize(50);
```

Batch writing

Batch writing can be enabled on the login with the `useBatchWriting()` method. Batch writing allows for groups of insert/update/delete statements to be sent to the database in a single batch, instead of one at a time. This can be a huge performance benefit. TopLink supports batch writing for selected databases and for JDBC 2.0 batch compliant drivers in JDK 1.2.

Some JDBC 2.0 drivers do not support batch writing. TopLink can be configured to support batch writing directly with the `dontUseJDBCBatchWriting()` method.

For more information, see Chapter 6, "Performance Optimization".

Example 1–8 Batch writing

```
project.getLogin().useBatchWriting();
    project.getLogin().dontUseJDBCBatchWriting();
```

Data optimization

By default, TopLink optimizes data access from JDBC, through avoiding double conversion by accessing the data from JDBC in the format that the application requires. For example, `long`s are retrieved directly from JDBC instead of having the driver return a `BigDecimal` that TopLink would then have to convert into a `long`.

Dates are also accessed as strings and converted directly to the date or `Calendar` type used by the application. Some JDBC drivers cannot convert the data correctly themselves so this optimization may have to be disabled. For example, some of the WebLogic JDBC drivers cannot convert dates to strings in the correct format.

Oracle's JDBC drivers were found to lose precision on floats in certain cases.

Note: The problems mentioned here may have been fixed in more recent versions of the drivers. Please check your vendor documentation for relevant updates.

Cache isolation

By default, concurrency is optimized and the cache is not locked more than required during reads or writes. The default settings allow for virtual concurrent reading and writing and should never cause any problems. If the application uses no form of locking then the last unit of work to merge changes wins. This feature allows for the isolation level of changes to the cache to be configured for severe situations only. It is not recommended that the default isolation level be changed.

Isolation settings are

- `ConcurrentReadWrite`: default
- `SynchronizedWrite`: allow only a single unit of work to merge into the cache at once
- `SynchronizedReadOnWrite`: do not allow reading or other unit of work merge while a unit of work is merging

Manual transactions

Sybase JConnect 2.x had problems with the JDBC auto-commit being used for transactions. This could prevent the execution of some stored procedures.

The `handleTransactionsManuallyForSybaseJConnect()` method gives a workaround to this problem. This problem may have been fixed in more recent versions of Sybase JConnect.

External transactions and connection pooling

TopLink supports integration with an application server's JTS driver and connection pooling. This support is enabled on the login. For more information, see Chapter 2, "Developing Enterprise Applications".

Other database connections

By default, TopLink uses the JDBC 1.0 standard technique for loading a JDBC driver and connecting to a database. That is, TopLink first loads and initializes the class by calling `java.lang.Class.forName()`, then obtains a connection to the database by calling `java.sql.DriverManager.getConnection()`. Some drivers do not support this technique for connecting to a database. As a result, TopLink can be configured in several ways to support these drivers.

Direct connect drivers

Some drivers (for example, Castanet drivers) do not support using the `java.sql.DriverManager` to connect to a database. TopLink instantiates these drivers directly, using the driver's default constructor, and obtains a connection from the new instance. To configure TopLink to use this direct instantiation technique, use the `useDirectDriverConnect()` method.

Example 1–9 Using useDirectDriverConnect()

```
project.getLogin().useDirectDriverConnect("com.foo.barDriver", "jdbc:foo:",
"server");
```

Using JDBC 2.0 data sources

The JDBC 2.0 specification recommends using a Java Naming and Directory Interface (JNDI) naming service to acquire a connection to a database. TopLink supports acquiring a database connection in this fashion. To take advantage of this feature, construct and configure an instance of `oracle.toplink.jndi.JNDIConnector` and pass it to the project login object using the `setConnector()` method.

Example 1–10 Using JNDI

```
import oracle.toplink.sessions.*;
import oracle.toplink.jndi.*;
    javax.naming.Context context = new javax.naming.InitialContext();
    Connector connector = new JNDIConnector(context, "customerDB");
    project.getLogin().setConnector(connector);
```

Custom database connections

TopLink also allows you to develop your own class that TopLink can use to obtain a connection to a database. The class must implement the `oracle.toplink.sessions.Connector` interface. This requires the class to implement three methods:

- `java.lang.Object clone()` — The object must be "cloneable."
- `java.sql.Connection connect(java.util.Properties properties)` — This method receives a dictionary of properties (including the user name and password) and must return a valid connection to the appropriate database.
- `void toString(PrintWriter writer)` — This method is used to print out any helpful information on the TopLink log.

After this class is implemented, it can be instantiated and passed to the project login object, using the `setConnector()` method.

Example 1–11 Using the oracle.toplink.sessions.Connector interface

```
import oracle.toplink.sessions.*;
    Connector connector = new MyConnector();
    project.getLogin().setConnector(connector);
```

Building database logins: examples

The following examples illustrate database login.

Example 1–12 A simple login procedure that reads an XML deployment file generated from TopLink Mapping Workbench

```
import oracle.toplink.tools.workbench.*;
    import oracle.toplink.sessions.*;
Project project = XMLProjectReader.read("C:\TopLink\example.xml");
project.getLogin().setUserName("userid");
project.getLogin().setPassword("password");

DatabaseSession session = project.createDatabaseSession();session.login();
```

Example 1–13 A simple login procedure that uses a generated project

```
import oracle.toplink.sessions.*;
Project project = new ACMEProject();
project.getLogin().setUserName("userid");
project.getLogin().setPassword("password");
DatabaseSession session = project.createDatabaseSession();
session.login();
```

Example 1–14 A simple login procedure that builds a login directly

```
import oracle.toplink.sessions.*;
DatabaseLogin login = new DatabaseLogin();
login.useJConnectDriver();
login.setDatabaseURL("dbserver:5000:acme");
login.setUserName("userid");
login.setPassword("password");
Project project = new Project(login);
DatabaseSession session = project.createDatabaseSession();
session.login();
```

Reference

Table 1–2 summarizes the most common public methods for the `DatabaseLogin`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for the `DatabaseLogin`, see the TopLink JavaDocs.

Table 1–2 Elements for DatabaseLogin

Element	Default	Method Names
Construction methods	not applicable	`Project getLogin()`
User name and password	some drivers / databases may default to the OS login; most do not	`setUserName(String name)` `setPassword(String password)`
Database platform	generic JDBC	`usePlatform(DatabasePlatform platform)`
Standard JDBC-ODBC bridge *only*	not applicable	`setDataSourceName(String dataSourceName)`
Other JDBC 1.0 Drivers	not applicable	`setDriverClassName(String driverClassName)` `setConnectionString(String url)`
Performance	do not use	`useBatchWriting()`
	(use JDBC if using JDK 1.2)	`dontUseJDBCBatchWriting()` `useJDBCBatchWriting()`
Creator/table qualifiers	none	`setTableQualifier(String qualifier)`
Parameter binding	printing byte arrays inline	`useByteArrayBinding()`
	arrays	`useStreamsForBinding()`
	50	`useStringBinding(int size)`
	print inline	`bindAllParameters()`
	do not cache	`cacheAllStatements()`

Table 1–2 Elements for DatabaseLogin (Cont.)

Element	Default	Method Names
Sequence Number information	do not use native sequencing	`useNativeSequencing()`
	'SEQUENCE'	`setSequenceTableName(String name)`
	'SEQ_COUNT'	`setSequenceCounterFieldName(String name)`
	'SEQ_NAME'	`setSequenceNameFieldName(String name)`
	50	`setSequencePreallocationSize(int size)`
JTS / EJB	not pooled	`useExternalConnectionPooling()`
	JDBC transactions	`useExternalTransactionController()`

Using the query framework

The term *query framework* describes the mechanisms used to read and write to the database. There are three ways to access the database.

Session queries TopLink provides methods for the DatabaseSession class that read and write at the object level. Session queries are the simplest way to access the database.

Query objects TopLink provides query object classes that encapsulate the database operations. Query objects support more options than the session queries, allowing complex operations to be performed.

Custom SQL queries TopLink internally generates SQL strings to access the database. The application can also call SQL directly or use SQL to build query objects. Use of custom SQL is discouraged in favor of session queries and query objects, but applications can always use SQL to customize the TopLink session queries or call stored procedures.

Session queries

The `DatabaseSession` class provides direct support for reading and modifying the database by providing read, write, insert, update and delete operations. Each of these operations can be performed by calling the appropriate session method. The session queries are very easy to use and are flexible enough to perform most database operations.

The UnitOfWork class can also be used to modify objects. Using a UnitOfWork is the preferred and optimal approach when modifications to the database are being made. The UnitOfWork has been optimized to keep track of changes that are being made to objects using object change sets. This enhancement allows the application to access change sets describing modifications made to an object within the unit of work or through event modification. Now, applications can check if any changes occurred before deciding to commit or release the unit of work. The application checks for changes by sending the hasChanged() message to the unit of work. Any changes are committed to the database by calling the UnitOfWork's commit method.

The UnitOfWork's 'commit and merge' algorithm has also been optimized to improve performance.

Query objects

The application can create query objects to perform more complex querying criteria than the session queries allow, if required. An application can create query objects by instantiating the appropriate query object and providing it with querying criteria. These criteria can be Expression objects or raw SQL strings.

Query objects can be used in one of four ways:

- They can be executed directly by calling the executeQuery() method on the DatabaseSession.
- They can define new querying routines and add them to the session. These new session queries are named so that they can be executed by name in a session.
- They can change the default querying behavior for read or write operations. An application can customize how the session's queries operate by supplying query objects to the descriptor's query manager.
- They can change the default querying behavior for complex relationship mappings such as selection queries.

Custom SQL queries

An application can also execute raw SQL strings and stored procedure calls. This is useful for calling stored procedures in the database and for accessing raw data.

Custom SQL strings and stored procedure calls can be used in one of three ways.

- They can be executed directly using the `executeSelectingCall()` and `executeNonSelectingCall()` session methods
- They can be executed through data-level queries by calling the `executeQuery()` method on the `DatabaseSession`.
- They can change the default querying behavior for read or write operations. An application can customize how the session's queries operate by supplying custom SQL to the descriptor's query manager.
- They can change the default querying behavior for complex relationship mappings such as selection queries.

Database exceptions

If an error is encountered during a database operation, a TopLink exception of type `DatabaseException` is thrown. Interaction with the database should be performed within a try-catch block to catch these exceptions.

```
try {
    Vector employees = session.readAllObjects(Employee.class);
    } catch (DatabaseException exception) {
    // Handle exception
    }
```

Refer to *Oracle9iAS TopLink Troubleshooting* for more information on handling TopLink exceptions.

Write operations can also throw an `OptimisticLockException` on a write, update or delete operation if optimistic locking is enabled.

For information on optimistic locking, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Querying on an inheritance hierarchy

When querying on a class that is part of an inheritance hierarchy, the session checks the descriptor to determine the type of the class.

If the descriptor has been configured to read subclasses, which is the default, the query returns instances of the class and its subclasses.

If the descriptor has been configured not to read subclasses, the query returns only instances of the queried class. It does not return any instances of the subclasses.

If neither of these conditions apply, the class is a leaf class, and does not have any subclasses. The query returns instances of the queried class.

Querying on interfaces

TopLink supports querying on an interface

- If there is only a single implementor of the interface, the query returns an instance of the concrete class.
- If there are multiple implementors of the interfaces, the query returns instances of all of the implementing classes.

> **Note:** Descriptors must be defined for interfaces to allow querying on them.

Using session queries

Session queries enables you to read and write objects in a database.

The `Session` class and its subclasses, such as `DatabaseSession` and `UnitOfWork`, provide methods to retrieve objects stored in a database. These methods are called query methods, and allow queries to be made in terms of the object model rather than the relational model.

Reading objects from the database

The session provides the following methods to access the database:

- *read operation*: Use the `readObject()` methods to read a single object from the database.
- *read all operation*: Use the `readAllObjects()` methods to read multiple objects from the database.
- *refresh operation*: Use the `refreshObject()` method to refresh the object with data from the database.

When looking for a specific object, it is preferable to use the `readObject()` methods rather than the `readAllObjects()` method, because a read operation based on the primary key may be able to find an instance in the cache and avoid going to the database. A read all operation does not know how many objects are to be retrieved, so even if it finds matching objects in the cache, it goes to the database to find any others.

Read operation

The `readObject()` methods retrieve a single object from the database. The application must specify the class of object to read. If no object matching the criteria is found, null is returned.

For example, the simplest reading operation would be:

```
session.readObject(MyDomainObject.class);
```

This example returns the first instance of `MyDomainObject` found in the table used for `MyDomainObject`.

Querying for the first instance of a class is not very useful. TopLink provides the `Expression` class to specify querying parameters for a specific object.

Read all operation

The `readAllObjects()` methods retrieve a `Vector` of objects from the database. The application must specify the class to read. An expression can be supplied to provide query parameters to identify specific objects within the collection. If no objects matching the criteria are found, an empty `Vector` is returned.

The `readAllObjects()` method returns the objects unordered.

Refresh operation

The `refreshObject()` method causes TopLink to update the object in memory with any new data from the database. This operation refreshes any privately owned objects as well.

Example 1–15 A typical use of readObject() using an expression

```
import oracle.toplink.sessions.*;
import oracle.toplink.expressions.*;

// Use an expression to read in the Employee whose last name is Smith. Create an
expression using the Expression Builder and use it as the selection criterion of
the search.
Employee employee = (Employee) session.readObject(Employee.class, new
ExpressionBuilder().get("lastName").equal("Smith"));
```

Example 1–16 A typical use of readAllObjects() using an expression

```
// Returns a Vector of employees whose employee salary > 10000.
Vector employees = session.readAllObjects(Employee.class,new
ExpressionBuilder.get("salary").greaterThan(10000));
```

Using expression builder

Applications need a flexible way to specify which objects are to be retrieved by a read query. Specifying query parameters using SQL would require application programmers to deal with relational storage mechanisms rather than the object model. Also, querying using strings is static and inflexible.

TopLink provides a querying mechanism called an expression that allows queries based on the object model. TopLink translates these queries into SQL and converts the results of the queries into objects.

Expression support is provided by two public classes. The `Expression` class represents an expression, which can be anything from a single constant to a complex clause with boolean logic. Expressions can be manipulated, grouped together and integrated in very flexible ways. The `ExpressionBuilder` serves as the factory for constructing new expressions.

Expression components A simple expression normally consists of three parts:

- an attribute
- an operator
- a constant for comparison

The attribute represents a mapped attribute or query key of the persistent class. The operator is an expression method that implements some sort of boolean logic, such as `between`, `greaterThanEqual` or `like`. The constant refers to the value used to select the object.

In the code fragment

```
expressionBuilder.get("lastName").equal("Smith");
```

the attribute is `lastName`, the operator is `equal()` and the constant is the string "`Smith`". The `ExpressionBuilder` is a stand-in for the object(s) to be read from the database; in this case, employees.

Use expressions instead of SQL Using expressions to access the database has many advantages over using SQL.

- Expressions are easier to maintain because the database is abstracted. Changes to descriptors or database tables do not affect the querying structures in the application.
- Expressions enhance readability by standardizing the query interface so that it looks similar to traditional Java calling conventions. For example, to get the street name from the Address object of the Employee class, write:

  ```
  emp.getAddress().getStreet().equal("Meadowlands");
  ```

  To use an expression to get the street name of an employee's address from the database, write:

  ```
  emp.get("address").get("street")
  .equal("Meadowlands");
  ```
- Expressions allow read queries to transparently query between two classes that share a relationship. If these classes are stored in multiple tables in the database, TopLink automatically generates the appropriate join statements to return information from both tables.

- Using expressions to specify read queries simplifies complex operations. For example, the following Java code retrieves all Employees living on "Meadowlands Drive" whose salary is greater than $10,000:

```
ExpressionBuilder emp = new ExpressionBuilder();
    Expression exp = emp.get("address").get("street") .equal("Meadowlands
    Drive");
    Vector employees = session.readAllObjects (Employee.class,
    exp.and(emp.get("salary") .greaterThan(10000)));
```

  TopLink automatically generates the appropriate SQL from that code:

```
SELECT t0.VERSION, t0.ADDR_ID, t0.F_NAME, t0.EMP_ID, t0.L_NAME, t0.MANAGER_
ID, t0.END_DATE, t0.START_DATE, t0.GENDER, t0.START_TIME, t0.END_TIME,
t0.SALARY FROM EMPLOYEE t0, ADDRESS t1 WHERE (((t1.STREET = 'Meadowlands')
AND (t0.SALARY > 10000)) AND (t1.ADDRESS_ID = t0.ADDR_ID))
```

Boolean logic Expressions use standard boolean operators such as AND, OR and NOT. Multiple expressions can be combined to form more complex expressions. For example, the following code fragment queries for projects managed by a selected person, with a budget greater than or equal to $1,000,000.

```
ExpressionBuilder project = new ExpressionBuilder();
Expression hasRightLeader, bigBudget, complex;
Employee selectedEmp = someWindow.getSelectedEmployee();
hasRightLeader = project.get("teamLeader").equal(selectedEmp);
bigBudget = project.get("budget").greaterThanEqual(1000000);
complex = hasRightLeader.and(bigBudget);
Vector projects = session.readAllObjects(Project.class, complex);
```

Functions TopLink supports a wide variety of database functions and operators, including `like()`, `notLike()`, `toUpperCase()`, `toLowerCase()`, `toDate()`, `rightPad()` and so on. Database functions allow you to define more flexible queries. For example, the following code fragment would match "SMITH", "Smith" and "smithers":

```
emp.get("lastName").toUpperCase().like("SM%")
```

Most functions are accessed through methods such as `toUpperCase` on the `Expression` class, but mathematical methods are accessed through the `ExpressionMath` class. This avoids over-complicating the Expression class with too many functions, while supporting mathematical functions similar to Java's java.lang.Math. For example:

```
ExpressionMath.abs(ExpressionMath.subtract(emp.get("salary"),emp.get("spouse").g
et("salary")).greaterThan(10000)
```

You may want to use a function in your database that TopLink does not support directly. For simple functions, use the getFunction() operation, which treats its argument as the name of a unary function and applies it. For example, the expression

```
emp.get("lastName").getFunction("FOO").equal(42)
```

would produce the SQL

```
SELECT . . . WHERE FOO(EMP.LASTNAME) = 42
```

You can also create more complex functions and add them to TopLink. See "Platform and user-defined functions" on page 1-36.

Expressions for one-to-one and aggregate object relationships Expressions can also use an attribute that has a one-to-one relationship with another persistent class. A one-to-one relation translates naturally into an SQL join that returns a single row. For example, to access fields from an employee's address:

```
emp.get("address").get("country").like("S%")
```

This example corresponds to joining the EMPLOYEE table to the ADDRESS table based on the "address" foreign key and checking for the country name. These relationships can be nested infinitely, so it is possible to ask for:

```
project.get("teamLeader").get("manager").get("manager").get("address").get("stre
et")
```

Expressions for one-to-many, many-to-many, direct collection and aggregate collection relationships More complex relationships can also be queried, but this introduces additional complications, because they do not map directly into joins that yield a single row per object.

TopLink allows queries across one-to-many and many-to-many relationships, using the `anyOf` operation. As its name suggests, this operation supports queries where any of the items on the "many" side of the relationship satisfy the query criteria.

For example:

```
emp.anyOf("managedEmployees").get("salary").lessThan(0);
```

returns employees where at least one of the employees who they manage (a one-to-many relationship) has a negative salary.

Similarly, we can query across a many-to-many relationship using:

```
emp.anyOf("projects").equal(someProject)
```

These queries translate into SQL and join the relevant tables, using a DISTINCT clause to remove duplicates. For example:

```
SELECT DISTINCT . . . FROM EMP t1, EMP t2 WHERE t2.MANAGER_ID = t1.EMP_ID AND
t2.SALARY < 0
```

Creating expressions with the Expression Builder `Expression` objects should always be created by calling `get()` or its related methods on an `Expression` or `ExpressionBuilder`. The `ExpressionBuilder` acts as a stand-in for the objects being queried. A query is constructed by sending it messages that correspond to the attributes of the objects. `ExpressionBuilder` objects are typically named according to the type of objects that they are used to query against.

Expression have been extended to support subqueries (SQL subselects) and parallel selects. A SubQuery can be created using an ExpressionBuilder and Parallel Selects allow for multiple heterogeneous expression builders to be used in defining a single query. In this way, joins are allowed to be specified for unrelated objects at the object level.

Parallel selects and sub-queries are discussed in more detail later in this chapter.

> **Note:** An instance of `ExpressionBuilder` is specific to a particular query. Do not attempt to build another query using the same builder, because it still has information related to the first query.

Example 1–17 A simple Expression built with the ExpressionBuilder

This example uses the query key "lastName" defined in the descriptor to reference the field name "L_NAME".

```
Expression expression = new ExpressionBuilder().get("lastName").equal("Young");
```

Example 1–18 An ExpressionBuilder example that uses the and() method

```
ExpressionBuilder emp = new ExpressionBuilder();
    Expression exp1, exp2;
    exp1 = emp.get("firstName").equal("Ken");
    exp2 = emp.get("lastName").equal("Young");
    return exp1.and(exp2);
```

Example 1–19 An ExpressionBuilder example that uses the notLike() method

```
Expression expression = new ExpressionBuilder().get("lastName").notLike("%ung");
```

Sub-selects and sub-queries Occasionally queries need to make comparisons based on the results of sub-queries. SQL supports this through sub-selects. Expressions provide the notion of sub-queries to support sub-selects.

Sub-queries allow for Report Queries to be included in comparisons inside expressions. A report query is the most SQL complete type of query in TopLink. It queries data at the object level based on a class and expression selection criteria. Report queries also allow for aggregation and group-bys.

Sub-queries allow for sophisticated expressions to be defined to query on aggregated values (counts, min, max) and unrelated objects (exists, in, comparisons). A sub-query is obtained through passing an instance of a report query to any expression comparison operation, or through using the `subQuery` operation on expression builder. The sub-query can have the same, or a different reference class and **must** use a different expression builder. Sub-queries can be nested or used in parallel. Sub-queries can also make use of custom SQL.

For expression comparison operations that accept a single value (`equal`, `greaterThan`, `lessThan`) the sub-query's result must return a single value. For expression comparison operations that accept a set of values (in, exists) the sub-query's result must return a set of values.

Example 1–20 A sub-query expression using a comparison and count operation

This example queries all employees that have more than 5 managed employees.

```
ExpressionBuilder emp = new ExpressionBuilder();
ExpressionBuilder managedEmp = new ExpressionBuilder();
ReportQuery subQuery =new ReportQuery(Employee.class, managedEmp);
subQuery.addCount();
subQuery.setSelectionCriteria(managedEmp.get("manager") .equal(emp));
Expression exp = emp.subQuery(subQuery).greaterThan(5);
```

Example 1–21 A sub-query expression using a comparison and max operation

This example queries the employee with the maximum salary in Ottawa.

```
ExpressionBuilder emp = new ExpressionBuilder();
ExpressionBuilder ottawaEmp = new ExpressionBuilder();
ReportQuery subQuery = new ReportQuery(Employee.class, ottawaEmp);
subQuery.addMax("salary");
subQuery.setSelectionCriteria(ottawaEmp.get("address").get("city").equal("Ottawa
"));
Expression exp =
emp.get("salary").equal(subQuery).and(emp.get("address").get("city").equal("Otta
wa"));
```

Example 1–22 A sub-query expression using a not exists operation

This example queries all employees that have no projects.

```
ExpressionBuilder emp = new ExpressionBuilder();
ExpressionBuilder proj = new ExpressionBuilder();
ReportQuery subQuery = new ReportQuery(Project.class, proj);
subQuery.addAttribute("id");
subQuery.setSelectionCriteria(proj.equal(emp.anyOf("projects")));
Expression exp = emp.notExists(subQuery);
```

Parallel expressions Occasionally queries need to make comparisons on unrelated objects. Expressions provide the notion of parallel expressions to support these types of queries. The concept of parallel queries is similar to sub-queries in that multiple expression builders are used. However a report query is not required.

The parallel expression must have its own expression builder and the constructor for expression builder that takes a `class` as an argument must be used. The class can be the same or different for the parallel expression and multiple parallel expressions can be used in a single query. Only one of the expression builders will be considered the primary expression builder for the query. This primary builder will make use of the zero argument expression constructor and its class will be obtained from the query.

Example 1–23 A parallel expression on two independent employees

This example queries all employees with the same last name as another employee of different gender (possible spouse).

```
ExpressionBuilder emp = new ExpressionBuilder();
ExpressionBuilder spouse = new ExpressionBuilder(Employee.class);
Expression exp =
emp.get("lastName").equal(spouse.get("lastName")).and(emp.get("gender").notEqual
(spouse.get("gender")).and(emp.notEqual(spouse));
```

Parameterized expressions, finders Expressions can also create comparisons based on variables instead of constants. This technique is useful for:

- customizing mappings
- creating re-usable queries
- defining EJB finders

In TopLink, a relationship mapping is very much like a query. It needs to know how to retrieve an object or collection of objects based on its current context. For example, a one-to-one mapping from `Employee` to `Address` needs to query the database for an address based on foreign key information from the table of the `Employee`. Each mapping contains a query, which in most cases is constructed automatically based on the information provided in the mapping. You can also specify these expressions yourself, using the mapping customization mechanisms described in the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

The difference from a regular query is that these are used to retrieve data for many different objects. TopLink allows these queries to be specified with arguments whose values are supplied each time the query is executed. We also need a way to refer directly to the potential values in the target database row without going through the object accessing mechanism.

The following two lower-level mechanisms are provided by the methods `getParameter()` and `getField()`.

Expression getParameter() Returns an expression representing a parameter to the query. The parameter is the fully qualified name of the field from the descriptor's row, or a generic name for the argument. This method is used to construct user defined queries with parameters or to construct the selection criteria for a mapping. It does not matter which `Expression` object this message is sent to, because all parameters are global to the current query.

Expression getField() Returns an expression representing a database field with the given name. Normally used to construct the selection criteria for a mapping. The argument is the fully qualified name of the field. This method must be sent to an expression that represents the table from which this field is derived. See also "Data-level queries" on page 1-37.

Example 1–24 The use of a parameterized expression in a mapping

This example builds a simple one-to-many mapping from class PolicyHolder to Policy. In this example, the SSN field of the POLICY table is a foreign key to the SSN field of the HOLDER table.

```
OneToManyMapping mapping = new OneToManyMapping();
mapping.setAttributeName("policies");
mapping.setGetMethodName("getPolicies");
mapping.setSetMethodName("setPolicies");
mapping.setReferenceClass(Policy.class);
```

```
// Build a custom expression here rather than using the defaults
ExpressionBuilder policy = new ExpressionBuilder();
mapping.setSelectionCriteria(policy.getField("POLICY.SSN")).equal(policy.getPara
meter("HOLDER.SSN")));
```

Example 1–25 Building a more complex Expression that can be used to perform a read query on a one-to-many mapping

```
ExpressionBuilder address = new ExpressionBuilder();
Expression exp = address.getField("ADDRESS.EMP_
ID").equal(address.getParameter("EMPLOYEE.EMP_ID"));
exp = exp.and(address.getField("ADDRESS.TYPE").equal(null));
```

Example 1–26 Using a parameterized expression in a custom query

The following example demonstrates how custom query is able to find an Employee if it is given the employee's first name.

```
ExpressionBuilder emp = new ExpressionBuilder();
Expression firstNameExpression;
firstNameExpression = emp.get("firstName").equal emp.getParameter("firstName"));
ReadObjectQuery query = new ReadObjectQuery();
query.setReferenceClass(Employee.class);
query.setSelectionCriteria(firstNameExpression);
query.addArgument("firstName");
Vector v = new Vector();
v.addElement("Sarah");
Employee e = (Employee) session.executeQuery(query, v);
```

Example 1–27 Using nested parameterized expressions

The following example demonstrates how custom query is able to find all employees living in the same city as a given employee.

```
ExpressionBuilder emp = new ExpressionBuilder();
Expression addressExpression;
addressExpression =
emp.get("address").get("city").equal(emp.getParameter("employee").get("address")
.get("city"));
ReadObjectQuery query = new ReadObjectQuery(Employee.class);
query.setName("findByCity");
query.setReferenceClass(Employee.class);
query.setSelectionCriteria(addressExpression);
query.addArgument("employee");
Vector v = new Vector();
v.addElement(employee);
Employee e = (Employee) session.executeQuery(query, v);
```

Platform and user-defined functions Different databases provide different functions and sometimes implement the same functions in different ways. For example, indicating that an order by clause is ascending might be ASC or ASCENDING. TopLink supports this by allowing functions and other operators that vary according to the relational database.

While most platform-specific operators already exist in TopLink, it is possible to add your own. For this, you must be aware of the `ExpressionOperator` class.

An `ExpressionOperator` has a selector and a Vector of strings. The selector is the identifier (id) by which users refer to the function. The strings are the constant strings that are used in printing this function. These strings are printed in alternation with the function arguments. In addition, you can specify whether the operator should be prefix or postfix. In a prefix operator, the first constant string prints before the first argument; in a postfix, it prints afterwards.

Example 1–28 Creating a new expression operator: the toUpperCase operator

```
ExpressionOperator toUpper = new ExpressionOperator();
toUpper.setSelector(-1);
Vector v = new Vector();
v.addElement("UPPER(");
v.addElement(")");
toUpper.printAs(v);
toUpper.bePrefix();

// To add this operator for all database
ExpressionOperator.addOperator(toUpper);
// To add to a specific platform
DatabasePlatform platform = session.getLogin().getPlatform();
platform.addOperator(toUpper);
```

Example 1–29 This example shows how the user-defined function can be accessed and queries for the firstname "foo" converted to upperCase "FOO"

```
ReadObjectQuery query = new ReadObjectQuery(Employee.class);
expression functionExpression = new
ExpressionBuilder().get("firstName").getFunction(ExpressionOperator.toUpper).equ
al("FOO");
query.setSelectionCriteria(functionExpression);
session.executeQuery(query);
```

Note: The `getFunction()` method can be called with a vector of arguments.

Data-level queries In TopLink, expressions are used for internal queries as well as for user-level queries. TopLink mappings build expressions internally and use them to retrieve database results. The expressions are, necessarily, at the data level rather than the object level, because they are part of what defines the object level.

It is also possible to build arbitrary data-level queries using TopLink. The main operations to be aware of are `getField()` and `getTable()`. You can call `getTable` to create a new table. You can either hold onto that table expression or subsequently call `getTable()` with the table name to fetch it.

Note that tables are specific to the particular expression to which `getTable()` was originally sent. The `getField()` message can be sent to expressions representing either tables or objects. In either case, the field must be part of a table represented by that object; otherwise, you will get an exception when executing the query.

In an object-level expression, you refer to attributes of objects, which may in turn refer to other objects. In a data-level expression, you refer to tables and their fields. You can also combine data-level and object-level expressions within a single query.

Example 1–30 Creating a data-level query

This example reads a many-to-many relationship using a link table and also checks an additional field in the link table. Note the combination of object-level and data-level queries, as we use the employee's manager as the basis for the data-level query. Also note the parameterization for the ID of the project.

```
ExpressionBuilder emp = new ExpressionBuilder();
Expression manager = emp.get("manager");
Expression linkTable = manager.getTable("PROJ_EMP");
Expression empToLink = emp.getField("EMPLOYEE.EMP_
ID").equal(linkTable.getField("PROJ_EMP.EMP_ID");
Expression projToLink = linkTable.getField("PROJ_EMP.PROJ_
ID").equal(emp.getParameter("PROJECT.PROJ_ID"));
Expression extra =
linkTable.getField("PROJ_EMP.TYPE").equal("W");
query.setSelectionCriteria((empToLink.and(projToLink)).and(extra));
```

Outer joins When querying, TopLink often uses joins to check values from other objects or other tables within the same object. This works well under most circumstances, but sometimes it is necessary to use a different type of join, known as an "outer join".

The most common circumstance is with a one-to-one relationship where one side of the relationship may not be present. For example, `Employee` objects may have an

Address object, but if the Address is unknown, it is null at the object level, and has a null foreign key at the database level.

Outer joins can also be used for one-to-many and many-to-many relationships for cases where the relationship is empty.

At the object level this works fine, but when issuing a read that traverses the relationship, objects may be missing. Consider the expression:

```
(emp.get("firstName").equal("Homer")).or(emp.get("address").
get("city").equal("Ottawa"))
```

In this case, employees with no address do not appear in the list, regardless of their first name. While non-intuitive at the object level, this is fundamental to the nature of relational databases and not easily changed. One way around the problem on some databases is to use an outer join. In this example, employees with no address show up in the list with null values in the result set for each column in the ADDRESS table, which gives the correct result. We specify that an outer join is to be used by using getAllowingNull() or anyOfAllowingNone() instead of get() or anyOf().

For example:

```
(emp.get("firstName").equal("Homer")).or(emp.getAllowingNull
("address").get("city").equal("Ottawa"))
```

Outer joins are useful but do have limitations. Support for them varies widely between databases and database drivers, and the syntax is not standardized. TopLink currently supports outer joins for Sybase, SQL Server, Oracle, DB2, Access, SQL Anywhere and the JDBC outer join syntax. Of these, only Oracle supports the outer join semantics in 'or' clauses. Outer joins are also used with ordering (see "Ordering for read all queries" on page 1-76) and for joining (see Chapter 6, "Performance Optimization").

Reference Table 1–3 and Table 1–4 summarize the most common public methods for ExpressionBuilder and Expression.

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for the ExpressionBuilder and Expression, see the TopLink JavaDocs.

Table 1–3 Elements for ExpressionBuilder

Element	ExpressionBuilder Method Names
Constructors	`ExpressionBuilder()` `ExpressionBuilder(Class aClass)`

Expression creation methods	`get(String queryKeyName)`
	`getAllowingNull(String queryKeyName)`
	`anyOf(String queryKeyName)`
	`anyOfAllowingNone(String queryKeyName)`
	`getField(String fieldName)`
	`in(ReportQuery subQuery)`

Table 1–4 Elements for Expression

Element	Expression Method Names
Constructors	Never use the `Expression` constructors. Always use an `ExpressionBuilder` to create a new expression.

Expression operators	`equal(Object object)`
	`notEqual(Object object)`
	`greaterThan(Object object)`
	`lessThan(Object object)`
	`isNull()`
	`notNull()`

Logical operators	`and(Expression theExpression)`
	`not()`
	`or(Expression theExpression)`

Key word searching	`equalsIgnoreCase(String theValue)` `likeIgnoreCase(String theValue)`
Aggregate functions (for use with report query)	`minimum()` `maximum()`

Relationship operators	`anyOf(String queryKeyName)`
	`anyOfAllowingNone(String queryKeyName)`
	`get(String queryKeyName)`
	`getAllowingNull(String queryKeyName)`
	`getField(String fieldName)`

Using query by example

TopLink's advanced expression framework queries can now be defined through providing example instances of the application's object model. This allows for highly dynamic queries to be easily defined through using the application's own object model. Query forms can be rapidly built through wiring an object model instance to the query form and passing the instance directly to the TopLink query. Query by example also provides a way for non-TopLink aware clients to define dynamic queries.

Query by example builds an expression from an instance by comparing each attribute to the current value in the instance. All types of direct mappings are supported as well as most relationship mappings, so the instance's related objects can also be queried. A policy object can also be used with the query. The policy can specify

- which attributes to compare
- which values to ignore (for example, do not compare null or empty strings)
- which operator to use (for example, LIKE, key word search) and if AND or OR should be used to combine the attribute comparisons.

Writing objects to the database

Although a `DatabaseSession` can write objects to the database directly, the `UnitOfWork` is the preferred approach when writing to the database in TopLink.

> **Note:** It is strongly recommended that `UnitOfWork` be used when writing to the database in TopLink, and that `writeObject()` not be used.

Writing a single object to the database

The `writeObject()` method should be called when a significant change to the object has occurred. It should also be called after the creation and initialization of new application objects so that the new objects are found in subsequent database queries.

The `writeObject()` method can be used on both new and existing instances stored in the database. It determines whether to perform an insert or an update by performing a *does exist* check. In essence, a 'does exist' check determines whether the object already exists in the database. If the object already exists, an update operation is performed. If it does not already exist, an insert operation is performed.

Privately owned objects are also written in the correct order to maintain referential integrity.

Writing all objects to the database

The application can write multiple objects using the `writeAllObjects()` method. It performs the same 'does exist' check as the `writeObject()` method and then performs the appropriate insert or update operations.

Adding new objects to the database

The `insertObject()` method should be called only when dealing with new objects. When using `insertObject()` instead of `writeObject()`, the 'does exist' check to the database is bypassed.

This method assumes that the object is a new instance and does not already exist in the database. If the object already exists in the database, an exception occurs when `insertObject()` is executed.

Modifying existing objects in the database

The `updateObject()` method should be called only when dealing with existing objects. When using `updateObject()` instead of `writeObject()`, the 'does exist' check to the database is bypassed.

This method assumes that the object already exists in the database. If the object does not already exist in the database, an exception occurs when `updateObject()` is executed.

Deleting objects in the database

To delete a TopLink object from its table, call the method `deleteObject()` and pass a reference to the object to delete.

An object must be loaded to be deleted. Any privately-owned data is also deleted when a `deleteObject()` operation is performed.

Writing objects: Examples

The following examples show how to implement `write` and `write all` operations in Java code.

Example 1–31 A typical use of writeObject()

```
//Create an instance of employee and write it to the database.
Employee susan = new Employee();
susan.setName("Susan");
...
//Initialize the susan object with all other instance variables.
session.writeObject(susan);
```

Example 1–32 A typical use of writeAllObjects()

```
// Read a Vector of all of the current employees in the database.
Vector employees = (Vector) session.readAllObjects(Employee.class);
...//Modify any employee data as necessary.
//Create a new employee and add it to the list of employees.
Employee susan = new Employee();
...
//Initialize the new instance of employee.
employees.add(susan);
//Write all employees to the database. The new instance of susan which is not
currently in the database will be inserted. All of the other employees which are
currently stored in the database will be updated.
session.writeAllObjects(employees);
```

Using transactions

A transaction is a set of database session operations that can either be committed or rolled back as a single operation.

If one operation in a transaction fails, all operations in the transaction fail. Transactions allow database operations to be performed in a controlled manner, in which the database is modified only when all transaction operations have been successful.

Transactions are closely related to the concept of a unit of work. If using a unit of work, transactions do not have to be used.

Transaction operations

TopLink provides the following methods to support transaction processing:

- `beginTransaction()` – marks the beginning of a transaction
- `commitTransaction()` – signals the end of a transaction, and is used to commit the set of transaction operations and modify the database
- `rollbackTransaction()` – once a transaction has been committed, there is no way to undo it; however, if an error occurs during a transaction, the `rollbackTransaction()` method can be used to undo the entire transaction set.

Nesting transactions

TopLink allows nested transactions but uses a single transaction in the database because JDBC does not support nested transactions. The inner transactions are counted and ignored.

Implementing a transaction in Java code

To add transaction processing to a set of database operations:

1. Call `beginTransaction()` at the start of the transaction set.
2. Specify a try-catch block that calls `rollbackTransaction()` if a database exception is thrown.
3. Call `commitTransaction()` at the end of the transaction set.

Example 1–33 This code updates all employee records. If an error occurs, the transaction is rolled back using rollbackTransaction().

```
/** Updates the group of employee records*/
void writeEmployees(Vector employees, Session session)
{
    Employee employee;
    Enumeration employeeEnumeration = employees.elements();
    try {
        session.beginTransaction();
        while (employeeEnumeration.hasMoreElements())
        {
            employee=(Employee) employeeEnumeration.nextElement();
            session.writeObject(employee);
```

```
            }
            session.commitTransaction();
        } catch (DatabaseException exception) {
        // A database exception has been thrown, indicating that at least one
        operation has failed. Roll back the Transaction if the application requires
        that all operations must succeed or all must fail.
            session.rollbackTransaction();
        }
}
```

Using units of work

A unit of work is a session that simplifies the transaction process and stores transaction information for its registered persistent objects. The unit of work enhances database commit performance by updating only the changed parts of an object.

Units of work are the preferred method of writing to a database in TopLink. The unit of work:

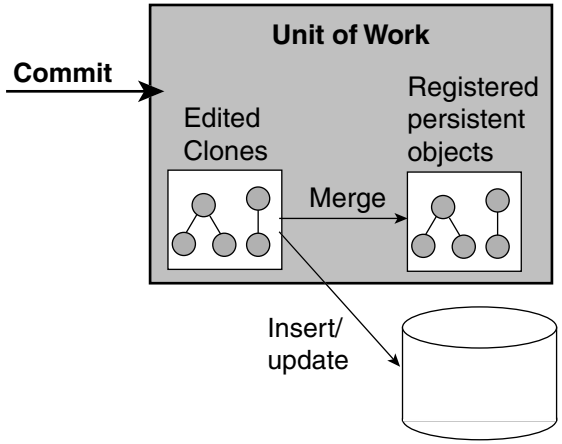
- Synchronizes changes to the databases and the object model
- Isolates edits of objects into their own transaction space
- Supports parallelism and nesting
- Sends a minimal amount of SQL to the database during the commit, by updating only the exact changes down to the field level
- Maintains referential integrity by ordering the inserts, updates, and deletes
- Resolves bi-directional references
- Avoids database deadlock through ordering table access

Understanding the unit of work

To use a unit of work, the application typically acquires an instance of `UnitOfWork` from the session and registers the persistent objects that are to change. The registering process returns clones that can be modified.

After changes are made to the clones, the application uses the `commit()` method to commit an entire transaction. The unit of work inserts new objects or updates changed objects in the database according to the changes made to the clones.

Figure 1–1 The life cycle of a unit of work



When writing the objects to the database:

- If an error occurs, a `DatabaseException` is thrown and the unit of work is rolled back to its original state.
- If no database error occurs, the original objects are updated with changes that were made to the clones.

Example 1–34 The typical life cycle of a unit of work

```
// The application reads a set of objects from the database.
Vector employees = session.readAllObjects(Employee.class);

// The application decides that a specific employee will be edited.
. . .
Employee employee = (Employee) employees.elementAt(index)

try {
    // Acquire a unit of work from the session.
    UnitOfWork uow = session.acquireUnitOfWork()
    // Register the object that is to be changed. The unit of work returns a
    clone of the object. We make the changes to the clone. The unit of work also
    makes a back-up copy of the original employee.
    Employee employeeClone = (Employee)uow.registerObject(employee)
    // We make changes to the employee clone by adding a new phoneNumber. If a
    new object is referred to by a clone, then it does not have to be
    registered. The unit of work determines that it is a new object at commit
```

```
    time.
    PhoneNumber newPhoneNumber = newPhoneNumber("cell","212","765-9002");
    employeeClone.addPhoneNumber(newPhoneNumber);
    // We commit the transaction. This causes the unit of work to compare the
    employeeClone with the back-up copy of the employee, begin a transaction,
    and update the database with the changes.
    // If all goes well, then the transaction is committed and the changes in
    employeeClone are merged into employee.
    // If there is an error updating the database, then the transaction is
    rolled back and the changes are not merged into the original employee
    object.
    uow.commit();
} catch (DatabaseException ex) {
// The commit has failed. The database was not changed. The unit of work should
be thrown away and application-specific action taken.
}
// After the commit, the unit of work is no longer valid. It should not be used
further.
```

Creating a unit of work

To create a unit of work for a given session, call the `acquireUnitOfWork()` method on the `DatabaseSession` class. The unit of work is valid until the application calls the `commit()` or `release()` methods.

Registering existing objects with a unit of work

Registering objects tells the unit of work that the application will change those objects. During registration, the unit of work creates and returns clones of the original objects given. All changes are made by the application on those clones. The original objects are left unchanged. If the `commit()` is successful, then the changes made to the clones are merged into the original objects.

You should use units of work to keep track of only those objects that are going to be changed. By registering objects that will not change, the unit of work is needlessly performing cloning and other processing.

The unit of work maintains object identity on the registered clones. If the same object is registered twice, the identical clone is returned.

When an object is read from the database using the unit of work, it is automatically registered with that unit of work, and therefore should not be re-registered.

Only root-level objects should be registered. New objects that are referred to by a clone do not have to be registered. At commit time, the unit of work determines that these are new objects, and takes appropriate action.

The unit of work has two methods to explicitly register objects:

- `registerObject(Object)` -- returns a clone of the object
- `registerAllObjects(Vector)` -- returns a `Vector` of clones

When objects are registered, the unit of work determines if they are new or existing (using the object's descriptor's "does exist" setting). If the objects are known to exist the `registerExistingObject()` method can be used to eliminate the need for the "does exist" check to be performed.

Reading objects using a unit of work

A unit of work is a `Session` and can be used for all database access during its lifetime. It uses the same methods to read from the database that a session uses, such as `readObject()` and `readAllObjects()`. These methods automatically register the objects read in the unit of work and return clones, so the `registerObject()` and `registerAllObjects()` methods do not have to be called.

Creating new objects in a unit of work

New objects can be included in a unit of work. Unless a registered clone points to them, the application must register these new objects so that they are written to the database at commit time.

The registration is done in the same way that you register other objects, by using the `registerObject()` call. If you do not register a newly created object, the `commit()` call does not write that object to the database, because the unit of work has no way of knowing that the new object exists.

Note: The registration of new objects still makes and returns a clone of the object. This clone must be used for further edits and the new object must be registered before being related to any other objects. An alternative method, `registerNewObject()`, can be used to register a new object without cloning. To avoid errors, new objects should be registered immediately after creation, or the `newInstance()` factory method can be used on the unit of work, which will instantiate and register a new instance of your object.

The order that `registerObject` is called on a new object does not affect the order in which objects are inserted. When the unit of work calculates its commit order, it uses the foreign key information in one-to-one and one-to-many mappings. If you are having constraint problems during insertion, make sure that your one-to-one mappings are defined correctly.

Writing objects using a unit of work

All updates and inserts on the database are done inside the call to the `UnitOfWork`'s `commit()` method. It is not valid to perform write, insert, and update operations on a unit of work. The `commit()` method updates the database with the changes to the cloned objects. Only those clones that have changed since they were registered are updated or inserted into the database.

If an error occurs when writing the objects to the database, a `DatabaseException` is thrown and the database is rolled back to its original state. If no database errors occur, the original objects are updated with the new values from the clones.

Successfully committing to the database ends the unit of work. The unit of work should not be used after a commit has been done.

Deleting objects through a unit of work

Deleting objects in a unit of work is done using the `deleteObject()` or `deleteAllObjects()` method. If an object being deleted has not been registered, then it is registered automatically.

When an object is deleted, its privately-owned parts are also deleted, because privately-owned parts cannot exist without their owner. At commit time, SQL is generated to delete the objects, taking database constraints into account.

If an object is deleted, then the object model must take the deletion of that object into account. References to the object being deleted must be set to null.

Example 1–35 Deleting an object through a unit of work

```
// Acquire a unit of work.
UnitOfWork uow = session.acquireUnitOfWork();

Project project = (Project) uow.readObject(Project.class);

Employee leader = project.getTeamLeader();

// Because we are deleting the Employee who is currently the team leader, we
```

```
must set the Project's teamLeader to be null. Otherwise, the object model will
be corrupted and the Project will be referring to a non-existent Employee.
// If the team leader is not set to null, then a QueryException will be thrown
during the merge. It is also likely that this would violate a database
constraint and a DatabaseException would be thrown during the commit.
project.setTeamLeader(null);

// Delete the leader employee at commit time.
uow.deleteObject(leader);

uow.commit();
```

Resuming a unit of work

Normally when a unit of work is committed, the clones of the registered objects become invalid. If another edit is started, the objects must be re-registered in a new unit of work and the new clones must be edited.

The unit of work also supports resuming, through the `commitAndResume()` and `commitAndResumeOnFailure()` methods. The changes in the unit of work are committed to the database; however, the unit of work is not invalidated. The same unit of work and clones of registered objects can continue to be used for subsequent edits and commits. If resume on failure is used and the unit of work commit fails, the unit of work can still be used and the commit re-tried.

Reverting a unit of work

Reverting a unit of work with the `revert()` method essentially puts the unit of work back in a state where all of the objects that were registered are still registered but no changes have been made yet.

In certain circumstances, an application may want to abandon the changes made to the clones in a unit of work, but does not want to abandon the unit of work. The method `revertAndResume()` exists for this purpose. The `revertAndResume()` method undoes all the changes made to the clones using the original objects as a guide. It also deregisters all new objects, and removes from the deletion set all of the objects for which `deleteObject(Object)` was called.

Executing queries from the unit of work

Like a session, a unit of work can execute queries using the `executeQuery()` method. The results of these queries are automatically registered in the unit of work and clones are returned to the caller.

Modify queries such as `InsertObjectQuery` or `UpdateObjectQuery` cannot be executed, because database modification is done only on commit.

Nested and parallel units of work

An application can have multiple units of work operating in parallel by calling `acquireUnitOfWork()` multiple times on the session. The units of work operate independently of one another and maintain their own cache.

The application can also nest units of work by calling `acquireUnitOfWork()` on the parent unit of work. This creates a child unit of work with its own cache.

The child unit of work should be committed or released before its parent. If a child unit of work commits, it updates the parent unit of work rather than the database. If the parent does not commit to the database, the changes made to the child are not updated in the database.

Inside a unit of work

The unit of work keeps track of original objects that are registered with it, the working copy clones and the back-up copy clones that it creates. The working copy clones are returned when an object is registered.

After the user changes the clones and commits the unit of work, the working copy clones are compared to the back-up copy clones. The changes are written to the database. The working copy clones are compared to the back-up copy clone (not to the original object) because another parallel unit of work may have changed the original object. Comparing to the back-up copy clones assures us that only the changes that were made in the current unit of work are written to the database and merged into the parent session's cache. The use of clones in the unit of work allows parallel units of work, which is an absolute requirement to build multi-user three-tier applications.

The creation of clones is highly optimized. When making clones, only mapped attributes are considered. The cloning process stops at indirection objects and continues only if the indirection objects are accessed. The cloning process is configurable using the descriptor's Copy Policy.

Advanced features

The Unit of Work offers a number of advanced features that enable you to optimize certain functions

Read-only classes

Within a unit of work, a class can be declared read-only. Declaring a class read-only tells the unit of work that instances of this class will never be modified. The unit of work can save time during registration and merge because instances of read-only classes do not require clones to be created or merged.

When an object is registered, the entire object tree is traversed and registered also. When a read-only object is encountered during the tree traversal, that branch of the tree is not traversed further. Therefore, any objects that are referred to by read-only objects are not registered either.

Read-only classes are normally reference data objects; that is, objects that are not changed in the current application.

An example of a reference data class would be the class `Country`. An `Address` can refer to a `Country` but the `Country` objects are created, modified, or deleted in another application. When modifying an `Address`, a `Country` object can be assigned to the `Address` where the `Country` object would have been chosen from a set of `Country` objects that are already stored in the database.

The user can set classes to be read-only for an individual unit of work immediately after it is acquired. The methods `addReadOnlyClass(Class)` or `addReadOnlyClasses(Vector)` can be used to change the set of read-only classes for a specific unit of work.

A default set of read-only classes can be established for the duration of the application by using the `Project` method `setDefaultReadOnlyClasses(Vector)`. All new units of work acquired after this call will have the Vector set of read-only classes.

Nested units of work have the same set or a super set of read-only classes as their parent. When a nested unit of work is acquired, it inherits the same set as its parent unit of work. If a class is declared read-only, then its subclasses must also be declared read-only.

Read-Only descriptors

TopLink's support for read-only classes within a unit of work extends to include descriptors (for information on Read-Only classes, see "Read-only classes" on page 1-51). When a class is declared as read-only, its descriptors are also flagged as read-only. In addition, you can flag a descriptor as read-only directly, either from within code or from the Mapping Workbench. The functionality is the same as for read-only classes, which improves performance by excluding read-only descriptors/classes from write operations such as inserts, updates, and deletes.

Descriptors can be flagged as read-only by calling the `setReadOnly()` method on the descriptor as follows:

```
descriptor.setReadOnly();
```

You can also flag a descriptor as read-only in the Mapping Workbench by checking the **Read Only** check box for a specific descriptor.

Always Conform Descriptors

TopLink's support for conforming queries in the unit of work can now be specified in the descriptors (for information on conforming queries, see "In-memory querying and unit of work conforming" on page 1-79). Conforming is specified at the query level. This enables the results of the query to conform with any changes to the object made within the unit of work including new objects, deleted objects and changed objects.

A descriptor can be directly flagged to always conform results in the unit of work so that all queries performed on this descriptor will, by default conform its results in the unit of work. This can be specified either within code or from the Mapping Workbench.

You can flag descriptors to always conform in the unit of work by calling the method on the descriptor as follows:

```
descriptor.setShouldAlwaysConformResultsInUnitOfWork(true);
```

You can also flag descriptors to always conform from the Mapping Workbench by checking the **Conform Results in Unit Of Work** check box for a descriptor.

Merging

When using the unit of work with a `ClientSession` in a three-tier application, objects are often returned from the client through some sort of serialization mechanism (for example, RMI or CORBA).

Note: TopLink also supports a remote session. In this case, the unit of work resides on the client, and TopLink handles the merging and replication issues.

The unit of work is expecting all changes to be made to the "working copy" clone that it returned when the original object was registered.

The changes to the object returned from the client must be propagated to the "working copy" clone of the unit of work before the unit of work is committed. The unit of work provides three methods, where each method takes a clone that was returned from the serialization mechanism and merges the changes into the unit of work's working copy clone:

- `mergeClone(Object)` – merges the clone and all of its privately-owned parts into the unit of work's working copy clone
- `deepMergeClone(Object)` – merges the clone and all of its parts into the unit of work's working copy clone
- `shallowMergeClone(Object)` – merges only the attributes that are mapped with direct mappings

Merge clone can be used with both existing and new objects. New objects can be merged only once within a unit of work, because they are not cached and may not have a primary key. If new objects are required to be merged twice, this can be done through the `setShouldNewObjectsBeCached()` method and ensuring that the objects have a valid primary key before being registered.

Validation

The unit of work validates object references when it commits. Objects registered in a unit of work should not reference objects that have not been registered in the unit of work. Doing this violates object transaction isolation and can lead to corrupting the session's cache. In some cases the application may wish to turn this validation off, or increase the amount of validation. This can be done through the `dontPerformValidation()` and `performFullValidation()` methods.

Troubleshooting the unit of work

When the unit of work detects an error during the merge, it throws a `QueryException` stating the invalid object and the reason that it is invalid. In this case, it may still be difficult for the application to figure out the problem, so the unit of work provides the `validateObjectSpace()` method to allow your application to pinpoint where the problem exists in the object model. The `validateObjectSpace()` method can be called at any time on the unit of work and provides the full stack of objects traversed to discover the invalid object.

Examples of units of work

The following examples show some typical units of work.

Example 1–36 Associating existing objects in a unit of work

```
// Get an employee read from the parent session of the unit of work.
Employee employee = (Employee)session.readObject(Employee.class)

// Acquire a unit of work.
UnitOfWork uow = session.acquireUnitOfWork();
Project project = (Project) uow.readObject(Project.class);

// When associating an existing object (read from the session) with a clone, we
must make sure we register the existing object and assign its clone into a unit
of work.

// INCORRECT: Cannot associate an existing object with a unit of work clone. A
QueryException will be thrown. project.setTeamLeader(employee);

// CORRECT: Instead register the existing object then associate the clone.
Employee employeeClone = (Employee)uow.registerObject(employee);
project.setTeamLeader(employeeClone);
uow.commit();
```

Example 1–37 Resolving issues involved in adding a new object when a bidirectional relationship exists

```
// Get an employee read from the parent session of the unit of work.
Employee manager = (Employee)session.readObject(Employee.class);

// Acquire a unit of work.
UnitOfWork uow = session.acquireUnitOfWork();

// Register the manager to get its clone
```

```
Employee managerClone = (Employee)uow.registerObject(manager);

// Create a new employee
Employee newEmployee = new Employee();
newEmployee.setFirstName("Spike");
newEmployee.setLastName("Robertson");

// INCORRECT: Should not be associating the new employee with the original
manager. This would cause a QueryException when TopLink detects this error
during the merge.
newEmployee.setManager(manager);

// CORRECT: associate the new object with the clone. Note that in this example,
the setManager method is maintaining the bidirectional managedEmployees
// relationship and adding the new employee to its managedEmployees. At commit
time, the unit of work will detect that this is a new object and will take the
appropriate action.
newEmployee.setManager(managerClone);

// INCORRECT: Do not register the newEmployee, as this would create two copies.
This would cause a QueryException when TopLink detects this error during the
merge.
// uow.registerObject(newEmployee);
// In the call to setManager, above, the managerClone's managedEmployees may not
have been maintained through the setManager method. If it were not the case, the
registerObject should have been called before the new employee was related to
the manager. If the developer was unsure if this was the case, the
registerNewObject method could be called to be sure that the newEmployee is
registered in the unit of work. The registerNewObject method registers the
object, but does not make a clone.
uow.registerNewObject(newEmployee);

// Commit the unit of work
uow.commit();
```

Reference

Table 1–5 summarizes the most common public methods for the `UnitOfWork`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for the `UnitOfWork`, see the TopLink JavaDocs.

Table 1–5 Elements for UnitOfWork

Element	Default	Method Names
Registering objects	not applicable	`registerObject(Object object)`
Nested units of work	not applicable	`acquireUnitOfWork()`
Query objects	not applicable	`executeQuery(DatabaseQuery query)`
Reading from the database	not applicable	`readAllObjects(Class domainClass, Expression expression)` `readObject(Class domainClass, Expression expression)`
Writing to the database	not applicable	`deleteObject(Object domainObject)`
Read-only classes (advanced)	all classes are read-write	`addReadOnlyClass(Class theClass)`
Merging clones	not applicable	`mergeClone(Object rmiClone)`

Working with locking policies

Locking policy is an important component of any multi-user TopLink application. When users share objects in an application, a locking policy ensures that two or more users do not attempt to modify the same object or its underlying data simultaneously. If the object is new, deleted, or changed, normal `insert`, `delete`, or `update` overrides the feature.

Many record locking strategies are employed by relational databases. TopLink includes support for the following locking policies:

Optimistic Lock All users have read access to the object. When a user attempts to write a change, the application checks to ensure the object has not changed since the last read.

Optimistic Read Lock Like an optimistic lock, the optimistic read lock checks to ensure the object has not changed since the last read when the user attempts to write a change. However, the optimistic read lock also forces a read of any related tables that contribute information to the object.

Pessimistic The first user who accesses an object with the purpose of updating locks the object until the update is complete. No other user can read or update the object until the first user releases the lock.

No locking The application does not verify that data is current.

> **Note:** When building a TopLink application you are most likely to use either optimistic locking or optimistic read locking as they are the safest and most efficient of these locking strategies.

Using optimistic lock

Optimistic locking, also known as write locking, allows unlimited read access to an object. However, a client can only write an object to the database if the object has not changed since the client last read it.

TopLink's support for optimistic record locking uses the descriptor, and can be applied in the following two ways:

- Version locking policies enforce optimistic locking by using version fields (or write lock fields) that are updated each time a record is written; a version field must be added to the table for this purpose
- Field locking policies do not require additional fields, but require `UnitOfWork` in order to be implemented.

> **Note:** In a three-tier application, if objects are edited on the client outside the context of a unit of work, then the write lock value must be stored in the object and passed to the client. If it is only stored in the cache on the server, then lock conflicts may be missed as other clients update the same cache.

Advantages and disadvantages

The advantages of optimistic locking are:

- It prevents users and applications from editing data that has been changed.
- Users can be notified immediately if there has been a locking violation when updating the object.
- It does not require the database resource to be locked up.
- It prevents database deadlocks.

The disadvantage of optimistic locking is

- It cannot prevent users and applications from editing data that is being changed; if two applications have the same data open, the first one to commit the changes succeeds while the other process fails.

Version locking policies

There are two types of version locking policies available in TopLink, `VersionLockingPolicy` and `TimestampLockingPolicy`. Each of these requires an additional field in the database to operate:

- For `VersionLockingPolicy`, add a numeric field to the database.
- For `TimestampLockingPolicy`, add a timestamp field to the database.

TopLink records the version as it reads an object from a table. When the client attempts to write the object, the version of the object is compared with the version in the table record. If the versions are the same, the updated object is written to the table, and the version of both the table record and the object are updated. If the versions are different, the write is disallowed and an error is raised.

The two version locking policies have different ways of writing the version fields back to the database:

- `VersionLockingPolicy` increments the value in the version field by one.
- `TimestampLockingPolicy` inserts a new timestamp into the row. The timestamp is configurable to get the time from the server or the local machine.

For both policies, the values of the write lock field can be stored in either the identity map or in a writable mapping within the object.

If the value is stored in the identity map, then by default an attribute mapping is not required for the version field. If the application does map the field, it must make the mappings read-only to allow TopLink to control writing the fields.

Field locking policies

TopLink support for field locking policies does not require any additional fields in the database. Field locking policy support includes:

- `AllFieldsLockingPolicy`
- `ChangedFieldsLockingPolicy`
- `SelectedFieldsLockingPolicy`

All of these policies compare the current values of certain mapped fields with their previous values. When using these policies, a `UnitOfWork` must be used for updating the database. Each policy handles its field comparisons in a specific way defined by the policy.

- Whenever an object using `AllFieldsLockingPolicy` is updated or deleted, all the fields in that table are compared in the where clause. If any value in that table has been changed since the object was read, the update or delete fails. This comparison is only on a per table basis. If an update is performed on an object that is mapped to multiple tables (including multiple table inheritance), only the changed table(s) appear in the where clause.
- Whenever an object using `ChangedFieldsLockingPolicy` is updated, only the modified fields are compared. This allows for multiple clients to modify different parts of the same row without failure. Using this policy, a delete compares only on the primary key.
- Whenever an object using `SelectedFieldsLockingPolicy` is updated or deleted, a list of selected fields is compared in the update statement. Updating these fields must be done by the application either manually or though an event.

Whenever any update fails because optimistic locking has been violated, an `OptimisticLockException` is thrown. This should be handled by the application when performing any database modification operations. The application must refresh the object and reapply its changes.

Java implementation of optimistic locking

Use the API to set optimistic locking in code. All of the API is on the descriptor:

- `useVersionLocking(String)` sets this descriptor to use version locking, and increments the value in the specified field name for every update or delete
- `useTimestampLocking(String)` sets this descriptor to use timestamp locking and writes the current server time in the specified field name for every update or delete
- `useChangedFieldsLocking()` tells this descriptor to compare only modified fields for an update or delete
- `useAllFieldsLocking()` tells this descriptor to compare every field for an update or delete
- `useSelectedFieldsLocking(Vector)` tells this descriptor to compare the field names specified in this vector of Strings for an update or delete

The following example illustrates how to implement optimistic locking using the VERSION field of EMPLOYEE table as the version number of the optimistic lock

```
descriptor.useVersionLocking("VERSION");
```

This code stores the optimistic locking value in the identity map. If the value should be stored in a non-read only mapping, then the code would be:

```
descriptor.useVersionLocking("VERSION", false);
```

The false indicates that the lock value is not stored in the cache but is stored in the object.

Advanced optimistic locking policies

TopLink includes the previously described optimistic locking policies, and all of these policies implement the OptimisticLockingPolicy interface. This interface is referenced throughout the TopLink code. It is possible to create more policies by implementing this interface and implementing the methods defined.

Using optimistic read lock

Optimistic read lock is an advanced type of optimistic lock that not only checks the version of the object, but also forces optimistic lock checking on an unchanged object by issuing an SQL "UPDATE … SET VERSION = ? WHERE … VERSION = ?" statement to the database. Optimistic read locking also allows modification of version field along with optimistic lock checking. An optimistic lock exception is thrown if the "VERSION" field has changed.

This feature is supported in UnitOfWork API as follows:

```
UnitOfWork.forceUpdateToVersionField(Object cloneFromUOW, boolean
shouldModifyVersionField)
UnitOfWork.removeForceUpdateToVersionField(Object cloneFromUOW);
```

This feature can only be used on objects that implement a version locking policy or timestamp locking policy. When an object that implements a version locking policy is updated, the version value is incremented or set to the current timestamp. For more information on version locking policies, see "Version locking policies" on page 1-59.

When is an object considered changed? UnitOfWork considers an object changed when its direct-to-field mapping's attribute or aggregate object mapping's attribute is modified. If an object is added to or removed from the relationship of the source object, or an object in the relationship is changed, UnitOfWork does not consider

this a changed in the source object and does not check optimistic locking for the source object when it commits.

Working with version fields

Optimistic read lock enables a UnitOfWork to either force an update to the version or leave the version without an update using the `forceUpdateToVersionField function` as follows:

```
UnitOfWork.forceUpdateToVersionField(cloneObject, true|false);
```

Using the true switch causes the version to be incremented, while the false switch leaves the version non-incremented. Whether or not the version should be incremented depends on the circumstances.

Leaving the version field unmodified Leave the version unmodified when the application logic depends on an unchanged object in the current application but the object may have changed in another application. Forcing optimistic lock checking on the object guarantees the validity of data committed in the current application.

TopLink-generated SQL for this feature typically follows the format "UPDATE … SET VERSION = 10 WHERE … VERSION = 10".

Example 1–38 Optimistic lock leaving the version field unmodified

In this example, a thread is calculating a mortgage rate based on the current interest rate (the "mortgage rate" thread). If the interest rate used by this thread is adjusted by another thread (the "interest rate" thread) while the calculation is happening. the calculation becomes invalid, because the mortgage rate thread does not take into account the changes made by the interest rate thread. To avoid this, the mortgage rate thread forces optimistic lock checking on the interest rate to guarantee a valid calculation.

The following code calculates the mortgage rate:

```
try {
    UnitOfWork uow = session.acquireUnitOfWork();
    MortgageRate cloneMortgageRate = (MortgageRate)
    uow.registerObject(mortgageRate);
    InterestRate cloneInterestRate = (InterestRate)
    uow.registerObject(interestRate);
    cloneMortgageRate.setRate(cloneInterestRate
    .getRate() - cloneMortgageRate.getDiscount());
    /* Force optimistic lock checking on interestRate to guarantee a valid
    calculation, but with no version update*/
    uow.forceUpdateToVersionField(cloneInterestRate, false);
```

```
    uow.commit();
}(OptimisticLockException exception) {
    /* Refresh the out-of-date object */
    session.refreshObject(exception.getObject());
    /* Retry… */
}
```

This code adjusts the interest rate:

```
try {
    UnitOfWork uow = session.acquireUnitOfWork();
    InterestRate cloneInterestRate = (InterestRate)
    uow.registerObject(interestRate);
    cloneInterestRate.setRate(cloneInterestRate
    .getRate() + 0.005);
    uow.commit();
}(OptimisticLockException exception) {
    /* Refresh out-of-date object */
    session.refreshObject(exception.getObject());
    /* Retry… */
}
```

Modifying the version field This feature is applied in situation where application requires marking an unchanged object as changed when it modifies the object's relationship.

TopLink-generated SQL for this feature typically follows the format "UPDATE … SET VERSION = 11 WHERE … VERSION = 10".

Example 1–39 Optimistic lock modifying the version field

A thread (the "bill" thread) is calculating an invoice for a customer. If another thread (the "service" thread) adds a service to the same customer or modifies the current service, the bill thread must be informed so that the changes are reflected on the invoice. This is accomplished as follows:

- The service thread marks the customer object as changed.
- The bill thread forces optimistic lock checking on the customer object.

This code represents the service thread. It adds a service to the customer and updates the version:

```
try {
    UnitOfWork uow = session.acquireUnitOfWork();
    Customer cloneCustomer = (Customer uow.registerObject(customer);
    Service cloneService = (Service uow.registerObject(service);
    /* Add a service to customer */
```

```
    cloneService.setCustomer(cloneCustomer);
    cloneCustomer.getServices().add(cloneSerVice);
    /* Modify the customer version to inform other application that the customer
    has changed */
    uow.forceUpdateToVersionField(cloneCustomer, true);
    uow.commit();
}
(OptimisticLockException exception) {
    /* Refresh out-of-date object */
    session.refreshObject(exception.getObject());
    /* Retry… */
}
```

Notice that the service thread forces a version update. The following code represents the bill thread, and calculates a bill for the customer. Notice that it does not force an update to the version:

```
try {
UnitOfWork uow = session.acquireUnitOfWork();
Customer cloneCustomer = (Customer) uow.registerObject(customer);
Bill cloneBill = (Bill) uow.registerObject(new Bill());
cloneBill.setCustomer(cloneCustomer);
/* Calculate services' charge */
int total = 0;
for(Enumeration enum = cloneCustomer.getServices().elements();
enum.hasMoreElements();) {
    total += ((Service) enum.nextElement()).getCost();
}
cloneBill.setTotal(total);
/* Force optimistic lock checking on the customer to guarantee a valid
calculation */
uow.forceUpdateToVersionField(cloneCustomer, false);
uow.commit();
}(OptimisticLockException exception) {
    /* Refresh the customer and its privately owned parts */
    // session.refreshObject(cloneCustomer);
    /* If the customer's services are not private owned then use a
    ReadObjectQuery to refresh all parts */
    ReadObjectQuery query = new ReadObjectQuery(customer);
    /* Refresh the cache with the query's result and cascade refreshing to all
    parts including customer's services */
    query.refreshIdentityMapResult();
    query.cascadeAllParts();
    /* Refresh from the database */
    query.dontCheckCache();
```

```
    session.executeQuery(query);
    /* Retry… */
}
```

Pessimistic locking

Pessimistic locking means that objects are locked before they are edited, which ensures that only one client is editing the object at any given time.

Pessimistic locking differs from optimistic locking in that locking violations are detected at edit time, not commit time. The TopLink implementation of pessimistic locking uses database row-level locks. Depending on the database, a lock attempt on a locked row either fails or is blocked until the row is unlocked.

Pessimistic locking, unlike optimistic locking, prevents users from editing data that is being changed. While acquiring a pessimistic lock on an object, the object must be refreshed to reflect it's most recent state, but optimistic locking only requires refreshing objects when a lock violation has been detected. As a result, optimistic locking is typically more efficient.

For information on optimistic locking, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Advantages and disadvantages

The advantages of pessimistic locking are:

- Pessimistic locking can prevent users and applications from editing data that is being changed or has been changed.
- Users are notified immediately on locking violations.

The disadvantages of pessimistic locking are that it:

- is not fully supported by all databases.
- uses extra database resources.
- can cause deadlocks.
- can cause excessive locking.

> **Note:** TopLink uses database row-level locking to implement pessimistic locking. Although this is the standard way of implementing pessimistic locking in the database, not all databases support row-level locking functionality. Please consult your database documentation to see if row-level locking and the `SELECT ... FOR UPDATE [NO WAIT]` API is supported.

Pessimistic locks exist only for the duration of the current transaction. A database transaction must be held open from the point of the first lock request until the commit. When the transaction is committed or rolled back, all of the locks are released. When using the unit of work, a transaction is automatically started when the first lock is attempted, and committed or rolled back when the unit of work is committed or released. If you are not using the unit of work you must manually begin a transaction on the session.

> **Note:** Using pessimistic locking requires that TopLink maintains an open transaction and database locks for a longer period than if optimistic locking were used. This can lead to database deadlocks. Also, when using the `ServerSession`, it decreases the concurrency of connection pooling, which affects the overall scalability of your application.

TopLink offers two methods of locking, `WAIT` and `NO_WAIT`. When refreshing an object in `WAIT` mode, the transaction must wait until the lock on the object is free before obtaining a lock on that object. In `NO_WAIT` mode, an exception is thrown if the object is being locked.

Example 1–40 Using pessimistic locking within a unit of work with refresh and WAIT for lock

```
import oracle.toplink.sessions.*;
import oracle.toplink.queryframework.*;
...
UnitOfWork uow = session.acquireUnitOfWork();
Employee employee = (Employee) uow.readObject(Employee.class);

// Note: This will cause the unit of work to begin a transaction. In a 3-Tier
model this will also cause the ClientSession to acquire its write connection
from the ServerSession's pool.
uow.refreshAndLockObject(employee, ObjectLevelReadQuery.LOCK);
// Make changes to object
```

```
...
uow.commit();
...
```

Example 1–41 Using pessimistic locking within a unit of work with refresh and NO WAIT for the lock

```
import oracle.toplink.sessions.*;
import oracle.toplink.queryframework.*;
import oracle.toplink.exceptions.*;
...
UnitOfWork uow = session.acquireUnitOfWork();
Employee employee = (Employee) uow.readObject(Employee.class);

try {
    employee = (Employee)
        uow.refreshAndLockObject(employee,
            ObjectLevelReadQuery.LOCK_NOWAIT);
    } catch (DatabaseException dbe) {
    // Some databases throw an exception instead of returning nothing.
    employee = null;
    }
    if (employee == null) {
        // Lock could not be obtained
        uow.release();
        throw new Exception("Locking error.");
    } else {
        // Make changes to object
        ...
        uow.commit();
    }
    ...
```

Example 1–42 Using pessimistic locking within a unit of work with ReadObjectQuery and ReadAllQuery

```
import oracle.toplink.sessions.*;
import oracle.toplink.queryframework.*;

...
UnitOfWork uow = session.acquireUnitOfWork();

ReadObjectQuery query = new ReadObjectQuery();
query.setReferenceClass(Employee.class);
query.acquireLocks();
```

```
// or acquireLocksWithoutWaiting()query
.refreshIdentityMapResult();
Employee employee = (Employee) uow.executeQuery(query);

// Make changes to object
...

uow.commit();
...

UnitOfWork uow = session.acquireUnitOfWork();
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.setSelectionCriteria(new
ExpressionBuilder().get("salary").greaterThan(25000));
query.acquireLocks();
// or acquireLocksWithoutWaiting()query
.refreshIdentityMapResult();
// NOTE: the objects are registered when they are obtained by using unit of
work. TopLink will update all the changes to registered objects when unit of
work commit.
Vector employees = (Vector) uow.executeQuery(query);
    // Make changes to objects
    ...
    uow.commit();
...
```

Example 1–43 Using pessimistic locking with a Session using ReadAllQuery

```
import oracle.toplink.sessions.*;
import oracle.toplink.sessions.queryframework.*;
...
// It must begin a transaction or the lock request will throw an exception
session.beginTransaction();
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.setSelectionCriteria(new
ExpressionBuilder().get("salary").greaterThan(25000));
query.acquireLocks();
// or acquireLocksWithoutWaiting()query.refreshIdentityMapResult();
Vector employees = (Vector) session.executeQuery(query);
// Make changes to objects
...
// Update objects to reflect changes
for (Enumeration enum = employees.elements();
```

```
    employees.hasMoreElements(); {
    session.updateObject(enum.nextElement());
}
session.commitTransaction();
...
```

Reference

Table 1–6 summarizes the most common public methods for `Pessimistic Locking`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for `Pessimistic Locking`, see the TopLink JavaDocs.

Table 1–6 Elements for Pessimistic Locking

Element	Default	Method Names
Lock mode (for ObjectLevelRead Query)	No lock	`acquiredLocks()` `acquiredLocksWithoutWaiting()`
Refresh and lock (for Session)	not applicable	`refreshAndLockObject(Object object, short lockMode)`

Session event manager

The session event manager handles information about session events. Applications can register with the session event manager to receive session event data.

Session events

As with descriptor events, `DatabaseSessions`, `UnitsOfWork`, `ClientSessions`, `ServerSessions`, and `RemoteSessions` raise `SessionEvents` when most session operations are performed. These events are useful when debugging or when coordinating the actions of multiple sessions. For more information on descriptor events, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Objects can register as listeners for these events by implementing the `SessionEventListener` interface and registering with the `SessionEventManager` using `addListener()`. Alternatively, objects can subclass `SessionEventAdapter`

and override only the methods for events required by your application. Currently, there is no support in TopLink Mapping Workbench for session events.

Events supported by the `SessionEventManager` include:

- `PreExecuteQuery`: raised before the execution of every query on the Session
- `PostExecuteQuery: raised` after the execution of every query on the Session
- `PreBeginTransaction`: raised before a database transaction is started
- `PostBeginTransaction`: raised after a database transaction is started
- `PreCommitTransaction`: raised before a database transaction is committed
- `PostCommitTransaction`: raised after a database transaction is committed
- `PreRollbackTransaction`: raised before a database transaction is rolled back
- `PostRollbackTransaction`: raised after a database transaction is rolled back
- `PreLogin`: raised before the Session is initialized and the connections acquired.
- `PostLogin`: raised after the Session is initialized and the connections have been acquired.

The following are raised only on a `UnitOfWork`:

- `PostAcquireUnitOfWork`: raised after a `UnitOfWork` is acquired
- `PreCommitUnitOfWork`: raised before `UnitOfWork` is committed
- `PrepareUnitOfWork`: raised after the `UnitOfWork` has flushed its SQL but before it has committed its transaction
- `PostCommitUnitOfWork`: raised after `UnitOfWork` is committed
- `PreReleaseUnitOfWork`: raised on a `UnitOfWork` before it is released
- `PostReleaseUnitOfWork`: raised on a `UnitOfWork` after it is released
- `PostResumeUnitOfWork`: raised on a `unitOfWork` after resuming

The following are three-tier events and are raised only in client/server sessions:

- `PostAcquireClientSession`: raised after a client session has been acquired
- `PreReleaseClientSession`: raised before releasing a client session
- `PostReleaseClientSession`: raised after releasing a client session
- `PostConnect`: raised after a new connection is established with the database
- `PostAcquireConnection`: raised after a connection is acquired

- `PreReleaseConnection`: raised before a connection is released

The following are database access events:

- `OutputParametersDetected`: raised after a stored procedure call with output parameters is executed, allowing for a result set and output parameters to be retrieved from a single stored procedure
- `MoreRowsDetected`: raised when a `ReadObjectQuery` detects more than one row returned from the database; you may want to raise this as a possible error condition in your application

Using the session event manager: examples

The following examples show how to use the session event manager.

Example 1–44 Registering a listener with the SessionEventManager

```
public void addSessionEventListener(SessionEventListener listener)
{
    // Register specified listener to receive events from mySession
    mySession.getEventManager().addListener(listener);
}
```

Example 1–45 Using SessionEventAdapter to listen for specific session events

```
. . .SessionEventAdapter myAdapter = new SessionEventAdapter() {
        // Listen for PostCommitUnitOfWork events
        public void postCommitUnitOfWork(SessionEvent event) {
        // Call my handler routine
    unitOfWorkCommitted();
    }
};
mySession.getEventManager().addListener(myAdapter);
. . .
```

Reference

Table 1–7 summarizes the most common public methods for the `SessionEventManager` class and Table 1–8 the `SessionEvent` class:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for the `SessionEventManager` and the `SessionEvent` class, see the TopLink JavaDocs.

Table 1–7 Elements of the SessionEventManager

Element	Default	Method Names
Listener registration	not applicable	`addListener(SessionEventListener listener)` `removeListener(SessionEventListener listener)`

Table 1–8 Elements of the SessionEvent

Element	Default	Method Names
All events		`getSession()` `getEventCode()`
Query events		`getQuery()` `getResult()`
Output parameters event		`getProperty("call")`

Query objects

Applications normally query and modify the database using session methods such as `readObject()`, or unit of work methods such as `commit()`. Internally, these session methods simply create a query object, initialize it with the given parameters, and use it to access the database. Query objects are Java abstractions of SQL calls.

The application can also create custom query objects to use with the session or the descriptor's query manager. Custom query objects can be used to:

- Create new query operations
- Create named queries that are registered with the session
- Customize the session's default database operations, such as `readObject()` and `writeObject()`

These techniques are useful for improving application performance or for creating complex queries.

Query object components

TopLink uses query objects to store information about a database query. The query object stores the following information:

- Name of the query
- Class that the query accesses
- Arguments for the query

Query types

Read queries can be performed using the following query objects:

- `ReadAllQuery`: reads a collection of objects from the database
- `ReadObjectQuery`: reads a single object from the database
- `ReportQuery`: reads information about objects from the database

Write queries can be performed using the following query objects:

- `DeleteObjectQuery`: removes an object from the database
- `InsertObjectQuery`: inserts new objects into the database
- `UpdateObjectQuery`: updates existing objects in the database
- `WriteObjectQuery`: writes either a new or existing object to the database, determining whether to perform an insert or an update

Raw SQL can be performed using the following query objects:

- `ValueReadQuery`: return a single data value; can be used for querying the size of a cursored stream
- `DirectReadQuery`: return a collection of column values; can be used for direct collection queries
- `DataReadQuery`: execute a selecting raw SQL string
- `DataModifyQuery`: execute a non-selecting raw SQL string

Creating query objects

Query objects are created by instantiating the object and calling either the `setSelectionCriteria()`, `setSQLString()`, or `setCall()` method to describe how the query is performed. The `setSelectionCriteria()` method passes an expression to the query object, the `setSQLString()` method passes raw SQL to the query object, and the `setCall()` method passes a database call to the query object.

When the application calls `executeQuery()` to use a query object, it can pass arguments to the query object. The arguments describe which objects should be returned by the query. Arguments can be added to a query using `addArgument()`. The arguments must be added in the same order that they are passed into the `executeQuery()` method.

After initialization, the query object may be registered with the session using the `addQuery()` method. The query must be named when it is registered. Once registered, the application can execute the query using its name.

Executing queries

To execute a query, the `Session` method `executeQuery()` is used with optional arguments. This method is overloaded to provide support for up to three arguments or a vector of arguments.

Queries executed with the `executeQuery(Query)` method do not have to be registered with the session or descriptor.

Example 1–46 A named read query with two arguments

```
// Define two expressions that map to the first and last name of the employee.
ExpressionBuilder emp = new ExpressionBuilder();
firstNameExpression = emp.get("firstName").equal(emp.getParameter("firstName"));
lastNameExpression = emp.get("lastName").equal(emp.getParameter("lastName"));

// Create the appropriate query and add the arguments.
ReadObjectQuery query = new ReadObjectQuery();
query.setReferenceClass(Employee.class);
query.setSelectionCriteria(firstNameExpression.and(lastNameExpression));
query.addArgument("firstName");
query.addArgument("lastName");

// Add the query to the session.
session.addQuery("getEmployeeWithName", query);
```

```
// The query can now be executed by referencing its name and providing a first
and last name argument.
Employee employee = (Employee) session.executeQuery("getEmployeeWithName",
"Bob", "Smith");
```

Query timeout

TopLink supports setting timeout on query objects. A query timeout value can be set in seconds to force a hung or long executing query to abort after the specified time has elapsed. A `DatabaseException` is thrown following the timeout.

Example 1–47 Timeout on query objects

```
// Create the appropriate query and set timeout limits
ReadAddQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.setQueryTimeout(2);
try{
    Vector employees = (Vector)
    session.executeQuery(query);
} catch (DatabaseException ex) {
    // timeout occurs
}
```

Read query objects

TopLink provides two different query classes for reading objects from the database. `ReadAllQuery` and `ReadObjectQuery` objects return persistent classes from the database:

- All read query objects must call the `setReferenceClass()` method to specify what class should be read from the database.
- Read query objects can call `setSelectionCriteria()` to specify an `Expression` that gives the criteria for the read.

Example 1–48 A simple ReadAllQuery

```
// Returns a Vector of employees whose employee ID is > 100.
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.setSelectionCriteria(new ExpressionBuilder.get("id").greaterThan(100));
Vector employees = (Vector) session.executeQuery(query);
```

Example 1–49 A named ReadObjectQuery

```
// Create the appropriate query and add the arguments.
ReadObjectQuery query = new ReadObjectQuery();
query.setReferenceClass(Employee.class);

// Add the query to the session.
session.addQuery("getAnEmployee", query);

// Query for the first employee in the database.
Employee employee = (Employee) session.executeQuery("getAnEmployee");
```

Parameterized SQL

Parameterized SQL can be enabled on individual queries. This is done through the `bindAllParameters()` and `cacheStatement()` methods. This causes TopLink to use a prepared statement, binding all of the SQL parameters and caching the prepared statement. If this query is re-executed, the SQL prepare can be avoided (which can improve performance). For more information, see Chapter 6, "Performance Optimization".

Ordering for read all queries

After a `ReadAllQuery`, the resulting collection of objects can be ordered using the `addOrdering()`, `addAscendingOrdering()`, and `addDescendingOrdering()` methods. Attribute names or query keys and expressions can be used to order on.

Example 1–50 Providing ordering for a read all query using the lastName and firstName query keys in ascending order

```
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.addAscendingOrdering ("lastName");
query.addAscendingOrdering ("firstName");
Vector employees = (Vector) session.executeQuery(query);
```

Example 1–51 Providing ordering for a read all query using the street address, descending case-insensitive order of cities, and manager's last name

```
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
ExpressionBuilder emp = new ExpressionBuilder();
query.addOrdering (emp.getAllowingNull("address").get("street"));
query.addOrdering
(emp.getAllowingNull("address").get("city").toUpperCase().descending());
```

```
query.addOrdering(emp.getAllowingNull("manager").get("lastName"));
Vector employees = (Vector) session.executeQuery(query);
```

Note: The use of `getAllowingNull` to use an outer join for the address and manager relationships. If we did not do this, then employees without an address or manager would not appear in the list. For more information, see "Outer joins" on page 1-37.

Specifying the collection class

By default, a `ReadAllQuery` returns its result objects in a Vector. The results can be returned in any collection class that implements the Collection or Map interface. For more information, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide* .

Using cursoring for a ReadAllQuery

The `ReadAllQuery` class has a number of methods for cursored stream and scrollable cursor support. For more information, see "Cursored streams and scrollable cursors" on page 1-93.

Query optimization

TopLink supports both joining and batch reading as ways to optimize database reads. Using these techniques, you can dramatically decrease the number of times the database is accessed during a read operation. The methods `addJoinedAttribute()` and `addBatchReadAttribute()` are used to configure query optimization. See Chapter 6, "Performance Optimization" for more information.

Query return maximum rows

A maximum rows size can be set on any read query to limit the size of the result set so that, at most, the specified number of objects is returned. This can be used to catch queries that could return an excessive number of objects.

Example 1–52 Setting the maximum returned object size on read query

```
ReadAllQuery query = new ReadObjectQuery();
query.setReferenceClass(Employee.class);
query.setMaxRows(5);
Vector employees = (Vector) session.executeQuery(query);
```

Partial object reading

TopLink supports querying partial objects. Any read query can return just a subset of the object's attributes instantiated. This can improve read performance when the full object is not required.

Because the partial objects are not full objects, they cannot be cached or edited. In addition, TopLink does not automatically include the primary key information in the partially populated object, so it must be explicitly specified as a partial attribute if you want to re-query or edit the object.

The `addPartialAttribute()` method is used to configure partial object reading. For more information, see Chapter 6, "Performance Optimization".

Refreshing the identity map cache during a read query

A query can also use the `refreshIdentityMapResult()` method to force the identity map to refresh with the results of the query.

Example 1–53 Refreshing the result of a query in the identity map

```
ReadAllQuery query = new ReadObjectQuery();
    query.setReferenceClass(Employee.class);
    query.setSelectionCriteria(new
    ExpressionBuilder().get("lastName").equal("Smith"));
    query.refreshIdentityMapResult();
    Employee employee = (Employee) session.executeQuery(query);
```

Read query classes that refresh the identity map can also configure the refresh operations to cascade to the object's privately owned parts or all the object's parts. When the `refreshObject()` method is called on the session, it refreshes the object and all of its privately owned parts. When a read query is created and refreshing is used, only the object's attributes are refreshed; the privately owned parts are not refreshed. To make the read query also refresh the object's parts, the `cascadePrivateParts()` or `cascadeAllParts()` methods should be called. Normally, an object should not be refreshed without refreshing its privately owned parts because if its privately owned parts have changed on the database, the object is inconsistent within the database.

Example 1–54 Performing a refresh query that also refreshes the object's privately owned parts

```
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.refreshIdentityMapResult();
query.cascadePrivateParts();
Vector employees = (Vector) session.executeQuery(query);
```

In-memory querying and unit of work conforming

In-memory querying can be configured at the query level for both read object and read all queries.

Not all expression features are supported in memory. The following in-memory query features are supported:

- `checkCacheByPrimaryKey()` — default; if a read object query contains an expression that compares more than the primary key, a cache hit can still be obtained through processing the expression against the object in memory
- `checkCacheThenDatabase()` — any read object query can be configured to query the cache completely before resorting to accessing the database
- `checkCacheOnly()` — any read object or read all query can be configured to query only the cache and return the result from the cache without accessing the database
- `conformResultsInUnitOfWork()` — any read object or read all query within the context of a unit of work can be configured to conform the results with the changes to the object made within the unit of work; includes new objects, deleted objects and changed objects

In-memory querying enables you to perform queries on the cache rather than the database. In-memory querying supports the following relationships:

- one-to-many
- many to many
- aggregate collection
- direct collection

The following table identifies the supported options:

Table 1–9 Supported in-memory queries

Type	Supported	Unsupported
Comparators	equal(..)	like(..)
	notEqual(..)	
	lessThan(..)	
	lessThanOrEqual(..)	
	greaterThan(..)	
	greaterThanOrEqual(..)	
	between(...)	
	notBetween(...)	
	isNull()	
	notNull()	
	in(...)	
Logical operators	or(..)	
	and(..)	
Joining	get(..)	
	getAllowingNull(..)	
	anyOf(..)	
	anyOfAllowingNone(..)	

Note : The relationships themselves must be in memory for in memory traversal to work (that is, all value holders in memory should be triggered for in memory querying to work across relationships)

Conforming results in a unit of work

Query results can be conformed in the unit of work across one-to-many as well as a combination of one-to-one and one-to-many relationships. The following is an example of a query across two levels of relationships - one-to-many and one-to-one.

```
Expression exp =
bldr.anyOf("managedEmployees").get("address").get("city").equal("Perth");
```

Note: When relationships in a query use indirection for performance reasons, the use of in-memory querying requires that all valueholders be triggered so that the objects will be available in the cache.

Handling exceptions resulting from in-memory queries

In-memory queries may fail for a number of reasons, the most common being that the query expression is too complex to be executed in memory. Other reasons include untriggered valueholders where indirection is being used. All object models using indirection should first trigger valueholders before conforming on the relevant objects. When in-memory queries fail, they generate exceptions.

Exceptions thrown by the conform feature are masked by default. However, TopLink includes an API that allows for exceptions to be thrown rather than masked. The API is:

`uow.setShouldThrowConformExceptions(ARGUMENT)`
`ARGUMENT` is an integer with one of the following values:

`0` Do not throw conform exceptions (default)

`1` Throw only valueholder exceptions

`2` Throw all conform exceptions

Note: When building new applications, consider specifying that all conform exceptions should be thrown. This provides a more detailed feedback for unsuccessful in-memory queries.

Disabling the identity map cache update during a read query

You can disable the identity map cache update normally performed by a read query by calling the `dontMaintainCache()` method. This is typically done for performance reasons, such as reading objects that will not be needed later by the application.

Example 1–55 Disabling the identity map cache update so that large groups of objects can be quickly read from the database

This example demonstrates how code reads Employee objects from the database and writes the information to a flat file.

```
// Reads objects from the employee table and writes them to an employee file.
void writeEmployeeTableToFile(String filename, Session session)
{
    Vector employeeObjects;
    ReadAllQuery query = new ReadAllQuery();
    query.setReferenceClass(Employee.class);
    query.setSelectionCriteria(new
    ExpressionBuilder.get("id").greaterThan(100));
        query.dontMaintainCache();
        Vector employees = (Vector) session.executeQuery(query);
    // Write all of the employee data to a file.
    Employee.writeToFile(filename, employees);
    }
```

Internal query object caches

Read query objects can maintain an internal cache of the objects previously returned by the query. This internal cache is disabled by default. Use the `cacheQueryResults()` method to enable the internal cache.

Example 1–56 Caching the result of a query in the internal query object cache

```
ReadObjectQuery query = new ReadObjectQuery();
query.setReferenceClass(Employee.class);
query.cacheQueryResults();

// Will read from the database.
Employee employee = (Employee) session.executeQuery(query);

// Will not read from the database a second time; will read from the query
object's cache instead.
Employee employee = (Employee) session.executeQuery(query);
```

Write query objects

A write query can be executed using a `WriteObjectQuery` instance instead of using the `writeObject()` method of the session. Likewise, the `DeleteObjectQuery`, `UpdateObjectQuery` and `InsertObjectQuery` objects can be used instead of the respective `Session` methods.

Example 1–57 Writing an object to the database using a WriteObjectQuery object

```
WriteObjectQuery writeQuery = new WriteObjectQuery();
writeQuery.setObject(domainObject);
session.executeQuery(writeQuery);
```

Example 1–58 Using other write query objects using similar syntax

```
InsertObjectQuery insertQuery= new InsertObjectQuery();
insertQuery.setObject(domainObject2);
session.executeQuery(insertQuery);

UpdateObjectQuery updateQuery= new UpdateObjectQuery();
updateQuery.setObject(domainObject2);
session.executeQuery(updateQuery);

DeleteObjectQuery deleteQuery = new DeleteObjectQuery();
deleteQuery.setObject(domainObject2);
session.executeQuery(deleteQuery);
```

Non-cascading write queries

By default, all write queries also write all privately owned parts. To write the object without its privately owned parts, call the `dontCascadeParts()` method. This is useful for optimization if it is known that only the object's direct attributes have changed. It can also be used to resolve referential integrity dependencies when writing large groups of independent newly-created objects. This is not required if a unit of work is used, because the unit of work internally resolves referential integrity.

Example 1–59 Performing a non-cascading write query

```
// theEmployee is an existing employee read from the database.
theEmployee.setFirstName("Bob");
UpdateObjectQuery query = new UpdateObjectQuery();
query.setObject(theEmployee);
query.dontCascadeParts();
session.executeQuery(query);
```

Disabling the identity map cache during a write query

A write query can be configured not to update the identity map cache by calling the `dontMaintainCache()` method. This is typically done for performance reasons, such as inserting objects that will not be needed later by the application.

Example 1–60 Disabling the identity map so that large groups of objects can be quickly inserted into the database

The code reads all the objects from a flat file and writes new copies of the objects into a table.

```
// Reads objects from an employee file and writes them to the
employee table.
void createEmployeeTable(String filename, Session session)
{
    Vector employeeObjects;
    Enumeration employeeEnumeration;
    Employee employee;
    InsertObjectQuery query;
    // Read the employee data file.
    employeeObjects = Employee.parseFromFile(filename);
    employeeEnumeration = employeeObjects.elements();
    while (employeeEnumeration.hasMoreElements()) {
        employee = (Employee)
        employeeEnumeration.nextElement();
        query = new InsertObjectQuery();
        query.setObject(employee);
        query.dontMaintainCache();
        session.executeQuery(query);
    }
}
```

Using query objects to customize the default database operations

TopLink provides default querying behavior for each of the read and write operations. This default behavior is sufficient for most applications. If the application requires custom query behavior for a particular persistent class, it can provide its own query objects that are used when one of the database operations is performed. See the *Oracle9iAS TopLink Mapping Workbench Reference Guide*, for more information.

Creating custom query operations

Applications can define their own custom queries in addition to using the standard read and write operations. If the custom query is specific to a persistent class, it should be registered with that class' descriptor. If the custom query is not specific to a particular class, it should be registered with the session. Registered queries are then executed by calling one of the `executeQuery()` methods of `DatabaseSession` or `UnitOfWork`.

Using Query Redirectors

You can combine query redirectors with the TopLink query framework to perform very complex operations, including operations that might not otherwise be possible within the query framework. To create a redirector, implement the `oracle.toplink.queryframework.QueryRedirector` interface. The `Object invokeQuery(DatabaseQuery query, DatabaseRow arguments, Session session)` method is executed by the query mechanism, which then waits for appropriate results for the Query type to be returned. This method is invoked each time the query is executed.

TopLink provides one pre-implemented redirector, the `MethodBasedQueryRedirector`. To use this redirector, create a static `invoke method` on a class, and use the `setMethodName(String)` API to instruct the query on what method to invoke to get the results for the query.

Note: If the query is executed on a UnitOfWork, the results are registered with that UnitOfWork, so any objects retrieved in the invoke must come from the Session Cache.

Reference

Table 1–10 and Table 1–11 summarize the most common public methods for `query object`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for `query object`, see the TopLink JavaDocs.

Table 1–10 Elements for query objects

Element	Default	Method Names
Selection specification (one of these)	not applicable	`setSelectionCriteria(Expresssion expression)`
Parameterized SQL	dynamic SQL	`bindAllParameters()` `cacheStatement()`
Predefined queries	not applicable	`addArgument(String argumentName)`

Table 1–11 Elements specific to read query objects (ReadObjectQuery and ReadAllQuery)

Element	Default	Method Names
Read queries	not applicable	`setReferenceClass(Class aClass)`
Read queries – refreshing the identity map	do not refresh	`refreshIdentityMapResult()`

Read queries – in-memory querying	check cache by primary key	`checkCacheByExactPrimaryKey()`
		`checkCacheByPrimaryKey()`
		`checkCacheThenDatabase()`
		`checkCacheOnly()`
		`conformResultsInUnitOfWork()`
		`setCacheUsage(int usage)`

Read queries – pessimistic locking	do not lock	`acquireLocks()` `acquireLocksWithoutWaiting()`
Read queries – partial object reading	full objects	`addPartialAttribute(String attributeName)`
Read queries – query optimization	not optimized	`addBatchReadAttribute (StringattributeName)` `addJoinedAttribute(StringattributeName)`
ReadAllQuery– ordering		`addOrdering(Expression ordering)`

Query by example

Query by example allows for queries to be specified by providing sample instances of the persistent objects to be queried. Query by example is an intuitive form of expressing a query, but limited in the complexity of queries that can be defined.

To define a query by example, a Read Object or a Read All Query is provided with a sample persistent object instance and an optional Query By Example Policy. The sample instance contains the data to be queried on and the query by example policy contains optional configuration settings (such as which operators to use and which attributes to ignore or consider).

Defining a sample instance

To create a sample instance (example object), any valid constructor can be used. Only the attributes on which the query is to be based should be set. All other attributes must be not set, or set to null. A default set of values other than null is ignored; these include zero, empty string, and false. To configure other values to be ignored or to force attributes to be included, you must use a query by example policy.

Any attribute that uses a direct mapping can be queried on. One-to-one relationships can also be queried on, including nesting. However, other relationship mappings cannot be queried on. The query is composed using `AND` to tie the attribute comparisons together.

Example 1–61 Using query by example; this example queries the employee named Bob Smith

```
ReadObjectQuery query = new ReadObjectQuery();
Employee employee = new Employee();
employee.setFirstName("Bob");
employee.setLastName("Smith");
query.setExampleObject(employee);

Employee result = (Employee) session.executeQuery(query);
```

Example 1–62 Using query by example; this example queries across the employee's address

```
ReadAllQuery query = new ReadAllQuery();
Employee employee = new Employee();
Address address = new Address();
address.setCity("Ottawa");
employee.setAddress(address);
query.setExampleObject(employee);

Vector results = (Vector) session.executeQuery(query);
```

Defining a query by example policy

Providing a sample instance (example object) allows for a large set of queries to be defined, but is limited to using equals and only ignoring null and default primitive values. The query by example policy allows for a larger set of queries to be defined.

The query by example policy provides the following options:

- Usage of like or other operations per class type of the attribute values compared
- "Ignore set" of attribute values to not include in comparisons
- Forced inclusion of attributes even if the attribute value is in the ignore set
- Usage of `isNull` or `notNull` for attribute values

To specify a query by example policy, an instance of `QueryByExamplePolicy` is provided to the query.

Example 1–63 Query by example policy; this example uses like for Strings and includes the salary even if it is zero

```
ReadAllQuery query = new ReadAllQuery();
Employee employee = new Employee();
employee.setFirstName("B%");
employee.setLastName("S%");
employee.setSalary(0);
query.setExampleObject(employee);
QueryByExamplePolicy policy = new QueryByExamplePolicy();
policy.addSpecialOperation(String.class, "like");
policy.alwaysIncludeAttribute(Employee.class, "salary");
query.setQueryByExamplePolicy(policy);
Vector results = (Vector) session.executeQuery(query);
```

Example 1–64 Query by example policy; this example uses key words for Strings, and ignores -1

```
ReadAllQuery query = new ReadAllQuery();
Employee employee = new Employee();
employee.setFirstName("bob joe fred");
employee.setLastName("smith mc mac");
employee.setSalary(-1);
query.setExampleObject(employee);
QueryByExamplePolicy policy = new QueryByExamplePolicy();
policy.addSpecialOperation(String.class, "containsAnyKeyWords");
policy.excludeValue(-1);
query.setQueryByExamplePolicy(policy);
```

```
Vector results = (Vector) session.executeQuery(query);
```

Combining query by example with expressions

Query by example can be combined with expressions to gain added complexity in the breadth of queries that can be defined. This is done through giving the query both a sample instance (example object) and an expression.

Example 1–65 Combining query by example with expressions; this example uses an example object and an expression

```
ReadAllQuery query = new ReadAllQuery();
Employee employee = new Employee();
employee.setFirstName("Bob");
employee.setLastName("Smith");
query.setExampleObject(employee);
ExpressionBuilder builder = new ExpressionBuilder();
query.setSelectionCriteria(builder.get("salary"). between(100000,200000);
Vector results = (Vector) session.executeQuery(query);
```

Reference

Table 1–12 summarizes the most common public methods for `query by example`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for `query by example`, see the TopLink JavaDocs.

Table 1–12 Elements for QueryByExamplePolicy

Element	Default	Method Names
Special operations	not applicable	`addSpecialOperation(Class theClass, String operation)`
Forced inclusion	none	`alwaysIncludeAttribute(java.lang.Class exampleClass, java.lang.String attributeName)`
		`includeAllValues()`
Attribute exclusion	none	`excludeValue(Object value)`
		`excludeDefaultPrimitiveValues()`

Table 1–12 Elements for QueryByExamplePolicy (Cont.)

Element	Default	Method Names
Null equality		`setShouldUseEqualityForNulls(boolean flag)`

Report query

Report query provides developers with a way to access information on a set of objects instead of the objects themselves. It provides the ability to select disperse data from a set of objects and their related objects. Report query supports all database reporting functions and features. Although the report query returns data (not objects), it does allow for this data to be queried and specified at the object level.

The result of a `ReportQuery` is a collection of `ReportQueryResult` objects that are similar in structure and behavior to a `DatabaseRow` or a `Hashtable`.

Report query features:

- A subset of the object's attributes and its related object's attributes can be specified, allowing for querying of light-weight information
- Complex object-level expressions can be used for the selection criteria and ordering criteria
- Support for database aggregate functions: SUM, MIN, MAX, AVG, and COUNT
- Support for group by expressions
- Support for requesting the primary key attributes to be retrieved with each `ReportQueryResult`. This makes it easy to request the real object from a light-weight result

> **Note:** TopLink report queries do not support multiple references to the same attribute in a single result set.

Example 1–66 Using a report query to query reporting information on employees

This example reports the total and average salaries for Canadian employees grouped by their city.

```
import oracle.toplink.queryframework.*;
...
ExpressionBuilder emp = new ExpressionBuilder();
ReportQuery query = new ReportQuery(emp);
```

```
query.setReferenceClass(Employee.class);
query.addMaximum("max-salary", emp.get("salary"));
query.addAverage("average-salary", emp.get("salary"));
query.addAttribute("city", emp.get("address").get("city"));

query.setSelectionCriteria(emp.get("address"). get("country").equal("Canada"));
query.addOrdering(emp.get("address").get("city"));
query.addGrouping(emp.get("address").get("city"));
Vector reports = (Vector) session.executeQuery(query);
```

Reference

Table 1–13 summarizes the most common public methods for `report query`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for the `report query`, see the TopLink JavaDocs.

Table 1–13 Elements for report queries

Element	Default	Method Names
Adding items to select	nothing selected	addAttribute(**String itemName**) addAttribute(**String itemName, Expression attributeExpression**) addAverage(**String itemName**) addAverage(**String itemName, Expression attributeExpression**) addMaximum(**String itemName**) addMaximum(**String itemName, Expression attributeExpression**) addMinimum(**String itemName**) addMinimum(**String itemName, Expression attributeExpression**) addSum(**String itemName**) addSum(**String itemName, Expression attributeExpression**) addStandardDeviation(**String itemName**) addStandardDeviation(**String itemName, Expression attributeExpression**) addVariance(**String itemName**) addVariance(**String itemName, Expression attributeExpression**) addCount() addCount(**String itemName**) addCount(**String itemName, Expression attributeExpression**) addItem(**String itemName, Expression attributeExpression**) addFunctionItem(**String itemName, Expression attributeExpression, String functionName**)
Group by	not grouped	addGrouping(**String attributeName**) addGrouping(**Expression expression**)
Retrieving primary keys	not retrieved	retrievePrimaryKeys() dontRetrievePrimaryKeys() setShouldRetrievePrimaryKeys(**boolean shouldRetrievePrimaryKeys**)

Note: ReportQuery inherits from ReadAllQuery so it also supports most ReadAllQuery properties)

Cursored streams and scrollable cursors

Working with large collections of persistent objects usually reduces the performance of an application. The two main factors that affect the performance of large collections of persistent objects are:

- The time it takes to read the collections in from the database
- The large amount of memory needed to hold the collections in memory

TopLink provides the `CursoredStream and ScrollableCursor` classes as a means of dealing more efficiently with large collections returned from queries more efficiently.

`CursoredStream` is a TopLink version of the standard Java `InputStream` class and provides forward streaming across a query result of objects.

`ScrollableCursor` is a TopLink version of the `Iterator/ListIterator` interface from JDK 1.2. It provides both forward and backward scrolling when used with a JDBC 2.0 compliant driver. `ScrollableCursor` is best used in JDK 1.2 but can be used in JDK 1.1 and implement the Enumeration interface.

Java streams

Java streams are used to access files, devices, and collections as a sequence of objects. A stream monitors its internal position; it also provides methods for getting and putting objects at the current position and for advancing the position.

A stream can be thought of as a view of a collection. The collection can be a file, device, or a `Vector`. A stream provides access to a collection one element at a time, in sequence.

Streams provide access to objects one at a time, making it possible to implement stream classes in which the stream does not contain all of the objects of a collection at the same time. This is a very useful technique you can use to build TopLink applications, because TopLink applications often include queries that generate large which are time-consuming to collect.

Streams allow the query results to be retrieved from tables in smaller numbers as needed, resulting in a performance increase.

Supporting streams

Cursored stream support is provided by calling the `useCursoredStream()` method of the `ReadAllQuery` class.

```
CursoredStream stream;
    ReadAllQuery query = new ReadAllQuery();
    query.setReferenceClass(Employee.class);
    query.useCursoredStream();
    stream = (CursoredStream) session.executeQuery(query);
```

Instead of getting a `Vector` containing the results of the query, an instance of `CursoredStream` is returned.

Using cursored streams and scrollable cursors: examples

Consider the following two code fragments:

Example 1–67 Using a vector

```
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
Enumeration employeeEnumeration

Vector employees = (Vector) session.executeQuery(query);
employeeEnumeration = employee.elements();

while (employeeEnumeration.hasMoreElements())
{
Employee employee = (Employee) employeeEnumeration.nextElement();
employee.doSomeWork();
}
```

Example 1–68 Using a cursored stream

```
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.useCursoredStream();

CursoredStream stream = (CursoredStream) session.executeQuery(query);
while (! stream.atEnd())
{
    Employee employee = (Employee) stream.read();
    employee.doSomeWork();
    stream.releasePrevious();
```

```
}
stream.close();
```

The first code fragment returns a `Vector` that contains all the employee objects. If ACME has 10,000 employees, then the `Vector` contains references to 10,000 `Employee` objects.

The second code fragment returns a `CursoredStream` instance rather than a `Vector`. The `CursoredStream` collection appears to contain all 10,000 objects, but it initially contains a reference to only the first 10 `Employee` objects. It will retrieve the rest of the objects of the collection as they are needed. In many cases, the application never needs to read all the objects.

This results in a significant performance increase; most applications start up faster.

The `releasePrevious()` message is optional. This releases any previously read objects, which frees up system memory. Released read objects are only removed from the cursored stream storage; they are not released from the identity map.

Optimizing streams

The performance of CursoredStream objects can be customized by providing a *threshold* and *page size* to the `useCursoredStream(int, int)` method.

The threshold specifies the number of objects to initially read into the stream. The default threshold is 10.

The page size specifies the number of elements to be read into the stream when the threshold is reached. Larger page sizes result in faster overall performance, but can introduce delays into the application when each page has to be loaded. The default page size is 5.

A cursored stream used with the `dontMaintainCache()` option greatly improves performance when dealing with a batch-type operation. A batch operation performs simple operations on large numbers of objects and then discards the objects. Cursored streams create the required objects only as needed, and the `dontMaintainCache()` option ensures that these transient objects are not cached.

Java iterators

In JDK 1.2, an `Iterator` interface is defined for iterating over a collection. The `ListIterator` interface extends the `Iterator` interface to provide backward scrolling and positioning. `ScrollableCursor` implements the `ListIterator` interface to allow scrolling over a large collection result set from the database without requiring to read in all of the data. Scrollable cursors can traverse their

contents, both absolutely and relatively. To use the `ScrollableCursor` object, the underlying JDBC driver must be compatible with JDBC 2.0 specifications.

Supporting scrollable cursor

Scrollable cursored stream support is provided by calling the `useScrollableCursor()` or the `useScrollableCursor (int threshold)` method of the `ReadAllQuery` class.

```
ScrollableCursor cursor;
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.useScrollableCursor();
cursor = (ScrollableCursor) session.executeQuery(query);
```

Instead of getting a `Vector` containing the results of the query, an instance of `ScrollableCursor` is returned.

Example 1–69 Using a scrollable cursor

```
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.useScrollableCursor();
ScrollableCursor cursor = (ScrollableCursor) session.executeQuery(query);
while (cursor.hasNext())
{
    Employee employee = (Employee) cursor.next();
    employee.doSomeWork();
}
cursor.close();
```

Traversing scrollable cursors

The `relative(int i)` method advances the row number in relation to the current row by `i` rows. The `absolute(int i)` method places the cursor at an absolute row position, 1 being the first row. In addition to the `absolute(int i)` and `relative(int i)` methods, scrollable cursors provide several other means of moving through their contents.

Example 1–70 Traversing a scrollable cursor

```
// Traversing a scrollable cursor.
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.useScrollableCursor();
```

```
ScrollableCursor cursor = (ScrollableCursor) session.executeQuery(query);

if (cursor.isAfterLast()) {
    while (cursor.hasPrevious()) {
        System.out.println(cursor.previous().toString());
        }
    }
cursor.close();
```

The `hasPrevious()` and `previous()` methods are provided to traverse from the last row towards the first and retrieve the object from the row. The `afterLast()` method places the cursor after the last row in the result set. Therefore, the first call to `previous()` places the cursor at the last row and returns that object.

SQL and stored procedure call queries

TopLink supports generating SQL for all database operations and provides an expression framework that supports defining simple and complex queries at the object level. Occasionally, your application may require a very complex query using custom SQL or the use of a stored procedure on the database. TopLink allows for all database operations to be customized through SQL or stored procedure calls.

The customization of descriptor and mapping database operations is discussed in Chapter 3, "Working with Enterprise JavaBeans", Chapter 5, "SDK for XML and Non-relational Database Access", and Chapter 6, "Performance Optimization".

SQL Queries

You can provide an SQL string to any query instead of an expression. In this case, the SQL string must return all of the data required to build an instance of the class being queried. The SQL string can be a complex SQL query, or a stored procedure call. You can invoke SQL queries through the session read methods, or through a read query instance.

Example 1–71 A session read object query is used with custom SQL

```
Employee employee = (Employee) session.readObject(Employee.class, "SELECT * FROM
EMPLOYEE WHERE EMP_ID = 44");
```

Example 1–72 A read all query is used with SQL

```
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.setSQLString("EXEC PROC READ_ALL_EMPS");
Vector employees = (Vector) session.executeQuery(query);
```

Data-level queries

TopLink provides the following data-level queries that you can use to query or modify data (not objects) in the database:

- `DataReadQuery` -- used for reading rows of data
- `DirectReadQuery` -- used for reading a single column of data
- `ValueReadQuery` -- used for reading a single value of data
- `DataModifyQuery` -- used for modifying data

Example 1–73 A session method is used with custom SQL to query user and time information

```
Vector rows = session.executeSQL("SELECT USER, SYSDATE FROM DUAL");
```

Example 1–74 A data modify query is used with SQL to switch the database

```
DataModifyQuery query = new DataModifyQuery();
query.setSQLString("USE SALESDATABASE");
session.executeQuery(query);
```

Example 1–75 A direct read query is used with SQL to read all ids of employees

```
DirectReadQuery query = new DirectReadQuery();
query.setSQLString("SELECT EMP_ID FROM EMPLOYEE");
Vector ids = (Vector) session.executeQuery(query);
```

Stored procedure calls

You can provide a `StoredProcedureCall` object to any query instead of an expression or SQL string. The procedure must return all of the data required to build an instance of the class being queried.

Output parameters

Output parameters can be used to define a read object query if they return the correct fields to build the object. The `StoredProcedureCall` object allows for output parameters to be used. Output parameters allow for additional information to be returned from a stored procedure. Some databases do not support returning result sets from stored procedures, so output parameters are the only way to return data. Sybase and SQL Server do support returning result sets from stored procedures so output parameters are normally not required for these databases.

Example 1–76 Stored procedure call with an output parameter

```
StoredProcedureCall call = new StoredProcedureCall();
call.setProcedureName("CHECK_VALID_POSTAL_CODE");
call.addNamedArgument("POSTAL_CODE");
call.addNamedOutputArgument("IS_VALID");
ValueReadQuery query = new ValueReadQuery();
query.setCall(call);
query.addArgument("POSTAL_CODE");
Vector parameters = new Vector();
parameters.addElement("L5J1H5");
Number isValid = (Number) session.executeQuery(query,parameters);
```

Cursor output parameters

Oracle does not support returning a result set from a stored procedure, but does support returning a cursor as an output parameter. When using the Oracle JDBC drivers, you can configure a `StoredProcedureCall` object to pass a cursor to TopLink as a normal result set.

Example 1–77 Stored procedure with a cursored output parameter

```
StoredProcedureCall call = new StoredProcedureCall();
call.setProcedureName("READ_ALL_EMPLOYEES");
call.useNamedCursorOutputAsResultSet("RESULT_CURSOR");
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.setCall(call);
Vector employees = (Vector) session.executeQuery(query);
```

Output parameter event

You can use stored procedures for a TopLink operation that does not allow for output parameter to be returned. When the stored procedure returns an error code indicating that the application wants to check for an error condition, TopLink raises

the session event `OutputParametersDetected` to allow the application to process the output parameters.

Example 1–78 Stored procedure with reset set and output parameter error code

```
StoredProcedureCall call = new StoredProcedureCall();
call.setProcedureName("READ_EMPLOYEE");
call.addNamedArgument("EMP_ID");
call.addNamedOutputArgument("ERROR_CODE");
ReadObjectQuery query = new ReadObjectQuery();
query.setCall(call);
query.addArgument("EMP_ID");
ErrorCodeListener listener = new ErrorCodeListener();
session.getEventManager().addListener(listener);
Vector args = new Vector();
args.addElement(new Integer(44));
Employee employee = (Employee) session.executeQuery(query, args);
```

Reference

Table 1–14 summarizes the most common public methods for the `stored procedure call`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for the `stored procedure call`, see the TopLink JavaDocs.

Table 1–14 Elements for stored procedure call

Element	Default	Method Names
Selection specification (one of these)	not applicable	`setProcedureName(String name)`

Input parameters	same name	`addNamedArgument(String name)`
		`addNamedArgument(String dbName, String javaName)`
		`addNamedArgumentValue(String dbName, Object value)`
		`addUnnamedArgument(String javaName)`
		`addUnnamedArgumentValue(Object value)`

Table 1–14 Elements for stored procedure call (Cont.)

Element	Default	Method Names

Output parameters	same name	`addNamedInOutputArgument(String name)`
		`addNamedInOutputArgument(String dbName, String javaName, String javaName, Class type)`
		`addNamedInOutputArgumentValue(String dbName, Object value, String javaName, Class type)`
		`public void addUnnamedInOutputArgument(String inArgumentFieldName, String outArgumentFieldName, Class type)`
		`public void addunnamedInOutputArgumentValue(Object inArgumentValue, String outArgumentFieldName, Class type)`

Output parameters	same name	`addNamedOutputArgument(String name)`
		`addNamedOutputArgument(String dbName, String javaName)`
		`addNamedOutputArgument(String dbName, String javaName, Class javaType)`
		`addUnnamedOutputArgument(String javaName)`
		`public void addunnamedOutputArgument(String argumentFieldName, Class type)`

Cursor output parameters	not applicable	`useNamedCursorOutputAsResultSet(String argumentName)` `useUnnamedCursorOutputAsResultSet()`

2

Developing Enterprise Applications

An enterprise application is an application that is designed to provide services to a broad range of users across an entire business. This chapter describes how to develop enterprise applications using TopLink, and discusses

- Three-tier and enterprise applications
- Client and server sessions
- Remote sessions
- Session broker
- Java Transaction Service (JTS)
- TopLink support for Java Data Objects (JDO)
- Distributed Cache Synchronization

This chapter also illustrates some of the TopLink features that enable it to integrate with industry-leading enterprise application servers.

Three-tier and enterprise applications

Three-tier applications are an extension of the client server paradigm that separates an application into three tiers instead of two. These tiers include the client, the application server and the database server. This model allows for application logic to be performed on both the server and client tiers and is scalable to Internet deployment.

An enterprise application is one that integrates multiple heterogeneous systems. An enterprise application may need to integrate with multiple database servers, a legacy application or mainframe application. An enterprise application may also be required to support multiple heterogeneous clients such as RMI, HTML, XML,

CORBA, DCOM, or telephony. The three-tier model allows for complex enterprise applications to be built through integrating with other systems in the application server tier. There are many different types of enterprise architectures.

TopLink can be used in any enterprise architecture that makes use of Java. TopLink has direct support for multiple different enterprise architectures and application server features. TopLink is not an application server but provides application server components. TopLink can also be used in a Java client and a Java supporting database server.

TopLink is certified 100% pure Java and can be used in any Java VM including:

- all Java application servers
- Java supporting databases such as Oracle 9i and DB2 UDB
- Java-compatible browsers such as Netscape and Internet Explorer
- server Java platforms such as AS/400, OS/390, and UNIX

Table 2–1 lists the features that TopLink supports for various enterprise architectures. This table can be used to determine the relevant TopLink features for your applications architecture.

Table 2–1 TopLink's features for enterprise architectures

Architecture	TopLink Features
HTML, servlets, JPSs	■ Client Server sessions ■ Session Manager
RMI	■ Client Server sessions ■ Remote sessions ■ Session Manager
CORBA	■ Client Server sessions ■ Remote sessions ■ Session Manager
EJB Session Beans	■ Client Server sessions ■ Remote sessions ■ Session Manager ■ JTS and external connection pooling support

Table 2–1 TopLink's features for enterprise architectures (Cont.)

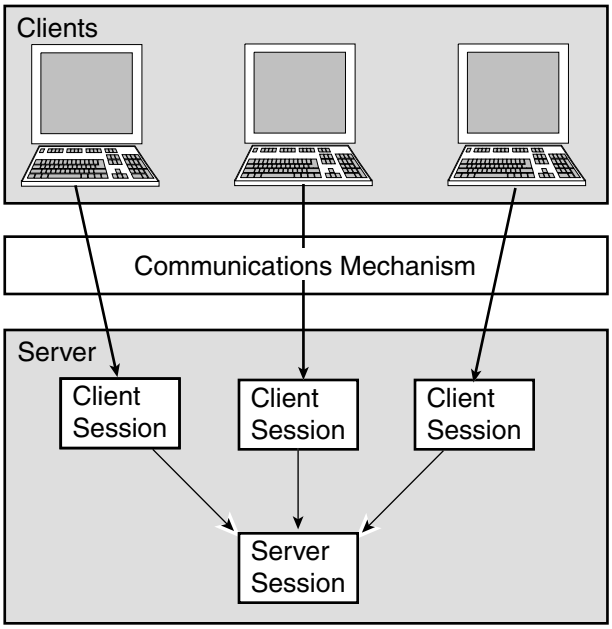
Architecture	TopLink Features
EJB Entity Beans	■ TopLink for Java Foundation Library bean-managed persistence ■ JTS and external connection pooling support ■ TopLink CMP for BEA WebLogic Server ■ TopLink CMP for IBM WebSphere Server
Java Transaction Service (JTS)	■ JTS and external connection pooling support
Multiple databases	■ Session Broker ■ JTS support
Multiple Application Servers (clustering)	■ Distributed cache synchronization
Java supporting databases	■ Oracle9i support
XML	■ TopLink SDK for XML
Enterprise Information System (EIS) access (non-relational/legacy databases)	■ TopLink SDK for EIS

Client and server sessions

Client and Server sessions provide the ability for multiple clients to share persistent resources. They provide a shared live object cache, read and write connection pooling, parameterized named queries and share descriptor metadata. Client and server sessions should be used in any application server architecture that supports shared memory and is required to support multiple clients.

Both the client and server sessions reside on the server. Clients can communicate through any communication mechanism to the application server. On the application server the client always communicates with a client session that in turn communicates to the database through the server session. Figure 2–1 shows how the client and server sessions are used. Client and server sessions are independent of the communications mechanism and should be used in architectures including HTML, Servlet, JSP, RMI, CORBA, DCOM and EJB.

Figure 2–1 Client and Server session usage



For a client to read objects from the database, it must acquire a `ClientSession` from the `ServerSession` or `Server` interface. This allows all client sessions to use the same shared object cache of the server session.

For a client to write objects to the database, it must acquire a `ClientSession` from the `ServerSession` or `Server` interface, and then acquire a `UnitOfWork` within that client session. The unit of work acts as an exclusive transactional object space. The unit of work ensures that any changes committed to the database through the unit of work are reflected in the server session's shared cache.

The server session or `Server` acts as the session manager for the three-tiered clients. The client session acts as a normal database session that is exclusive to each client or request.

For the most part, the client sessions are not used any differently than a normal TopLink database session. The client session supports all querying protocol that the database session supports.

Client sessions have two restrictions that are required to allow a shared object cache.

- All changes to the database must be done using a unit of work.
- All clients must be able to share a common database login for reading (different logins are supported for writing).

Users who have special security access (such as managers accessing salary information) cannot share the same cache as users who do not have access to that information. If multiple security levels exist, then a different server session must be used for each security level. Alternatively, non-shared database sessions could be used for each user with special security access.

Client sessions

A client session represents the dialog of one client with the server. The client session's lifecycle should mirror the lifecycle of the client. In a stateful three-tier model, the client session should exist until the client disconnects from the application server. In a stateless three-tier model, the client session should exist for the duration of one request of a client to the server. The client has exclusive access to the client session and should call the `release()` method on the client session object when it disconnects from the server. If notification of a disconnect cannot be guaranteed, the application server should time-out the connection to the client and force the client session to be released. If the client session garbage collects, it will automatically release itself.

Client sessions have many of the same properties as normal database sessions, but cannot use the following session properties:

transactions Client sessions should not explicitly begin transactions, but instead should leverage the TopLink unit of work.

schema creation Client sessions should not use the `SchemaManager`.

adding descriptors Client sessions cannot add descriptors.

write or delete Client sessions should not explicitly write or delete from the database. The client must acquire a unit of work (see Chapter 6, "Performance Optimization") to be able to modify the database.

Server sessions

The server session manages the client session, shared cache and connection pools. Although the server session is a TopLink session it should only be used to manage the servers client sessions. For this purpose the `Server` interface is provided. The `Server` interface does not implement the session API but only the public API required for the server session such as configuring connection pools and acquiring client sessions.

Servers can create new client sessions using the `acquireClientSession()` method.

Caching database information on the server

The data returned when a client reads an object is automatically cached on the server. This allows all client sessions to share a single cache stored in the server session's identity maps.

Ideally the `SoftCacheWeakIdentityMap` should be used. This identity map guarantees object identity. Because it uses weak references, it does not in itself impose memory requirements on the server. The `SoftCacheWeakIdentityMap` is available only if your VM supports the Java 2 API.

If the virtual machine (VM) being used does not implement the Java 2 API, then both the `FullIdentityMap` and `CacheIdentityMap` could be used. When using a full identity map, a reference is kept for all of the objects read in by all of the clients even after the reference is no longer needed. This imposes memory requirements on the server.

A possible solution to this problem is for the server system to periodically instruct TopLink to flush the cache. This can be done on a per instance or class basis, or for the identity map as a whole.

Another solution would be to use a cache identity map with a very large cache size. Objects that have been in the cache for a long period of time are eventually discarded. Note that this may lead to a loss of object identity. It is the responsibility of the server application to make sure that this does not occur by removing unnecessary references to objects in memory. Optimistic locking can also be used with a cache identity map to ensure that objects written to the database are not in an invalid state.

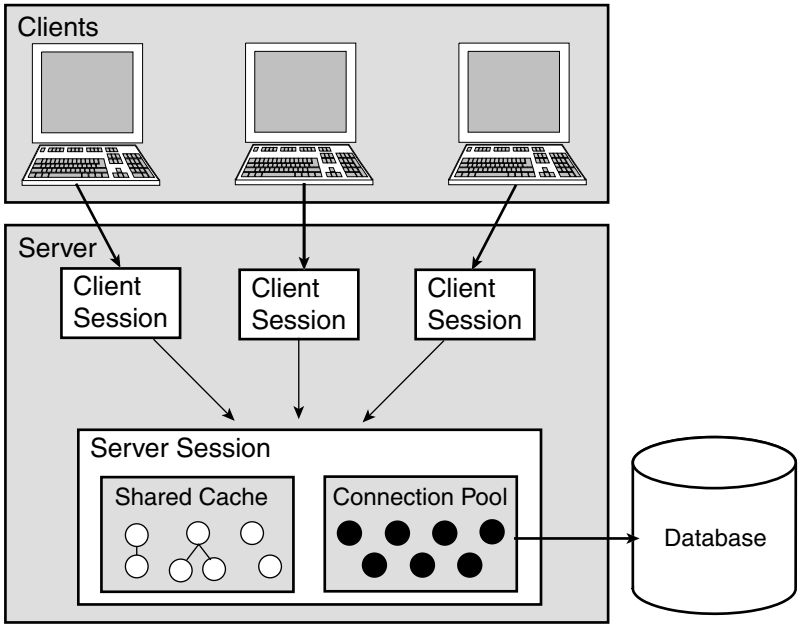
Providing client read access

Once the client acquires a client session, it can send read requests to the server. If the server can satisfy the read request with information from its object cache, it returns the information back to the client. If the server cannot satisfy the request with information from its cache, it reads from the database and stores the information in its cache. Subsequent request for that information returns information from the fast object cache instead of performing resource intensive database operations.

This server structure allows for all clients and client sessions to share the same object cache and the same database connection pool for reading. The server should deal with each client request in a separate thread so that the database connection pool can be used concurrently by multiple clients.

Figure 2–2 illustrates how multiple clients can read from the database using the server session.

Figure 2–2 Multiple client sessions reading the database using the server session



To read objects from the database:

1. Create a `ServerSession` object and call `login()` on it. This should be done only once, when the application server starts.

2. Acquire a `ClientSession` from the `ServerSession` by calling `acquireClientSession()`.
3. Execute read operations on the `ClientSession` object.

You should never use the `ServerSession` object for reading objects from the database.

Providing client write access

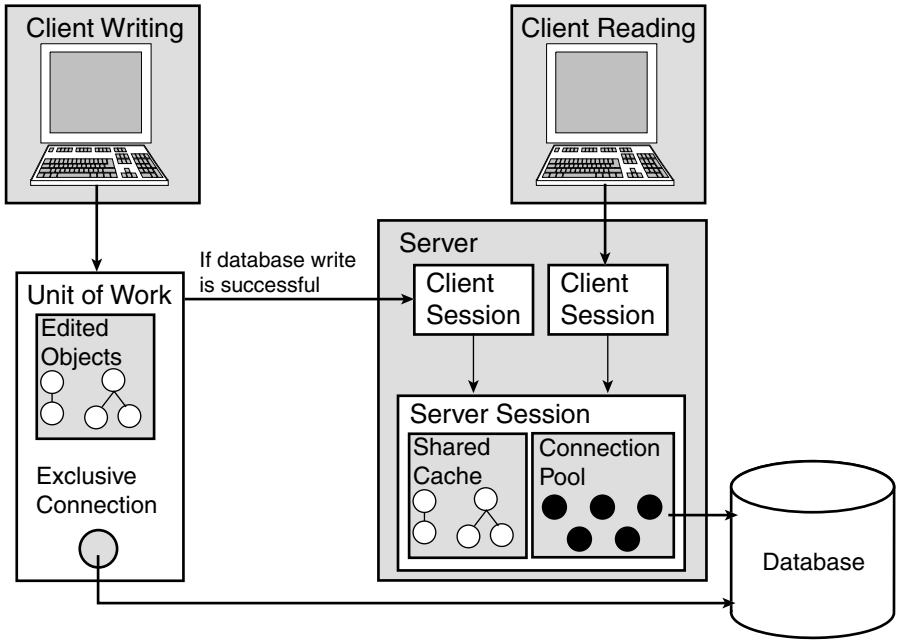
When the client wants to write to the database, it must acquire its own object transaction space. This is because the client and server sessions allow all clients to share the same object cache and the same objects (see Figure 2–3).

The client session disables all database modification methods so that objects cannot be written or deleted. The client must obtain a unit of work from the client session to perform database modification.

Caution: While client sessions are thread-safe, they should not be used to write across multiple threads. Multi-thread writes from the same client session can result in errors being thrown and a loss of data.

The unit of work ensures that objects are edited under a separate object transaction space. This allows clients to perform object transactions in parallel. Once completed, the unit of work performs the changes in the database and then merges all of the changes into the shared TopLink cache in the session to make the changes available to all other users. Refer to Chapter 1, "Working with Database Sessions" for more information on unit of work.

Figure 2–3 Writing with client sessions and server sessions



To write to the database:

1. Create a `ServerSession` object and call `login()` on it (this should be done only once, when the application server starts).
2. Call `acquireClientSession()` to acquire a `ClientSession` object from the `ServerSession`.
3. Acquire a `UnitOfWork` object from the `ClientSession` object. Refer to Chapter 6 for more information on unit of work.
4. Perform any updates that are required, then commit the `UnitOfWork`.

Concurrency

To have concurrent clients logged in at the same time, the server must spawn a dedicated thread of execution for each client. The RMI and CORBA application servers do this automatically. Dedicated threads enable each client to perform its desired work without having to wait for the completion of other clients. TopLink

ensures that these threads do not interfere with each other when they make changes to the identity map or perform database transactions.

TopLink addresses thread safety issues by using a concurrency manager for all of its critical components. The concurrency manager ensures that no two threads interfere with each other when altering critical data. Concurrency management is applied to crucial functions that include updating the cache when creating new objects, performing a transaction in the database, and accessing value holders.

Connection pooling

Connection pooling allows for the number of connections used by the server and client sessions to be managed and shared among multiple clients. This reduces the number of connections required by the application server, allowing for a larger number of clients to be supported.

Multiple connections can also be allocated for reading. Although a single connection can support multiple threads reading asynchronously, some JDBC drivers may perform better when multiple connections are allocated. If multiple connections are used for reading, TopLink balances the load across all of the connections using a least-busy algorithm.

By default, TopLink uses a connection pool to manage the connections between client and server sessions:

- The default write connection pool has a minimum of five connections and a maximum of 10.
- The default read connection pool has two connections.
- The minimum and maximum number of connections can be configured.

The default number of connections is fairly low to maintain compatibility with JDBC drivers that do not support many connections. A larger number of connections should be used for both reading and writing if supported by the JDBC driver.

Some JDBC drivers do not support concurrency so may require a thread to have exclusive access to a JDBC connection when reading. The server session should be configured to use exclusive read connection pooling in these cases.

The server session also supports multiple write connection pools and non-pooled connections. If your application server or JDBC driver also supports connection pooling, the server session can be configured to integrate with this connection pooling.

ServerSession connection options

The server session contains a pool of read connections and a pool of write connections that the client session may use. The number and behavior of each can be customized using the following `ServerSession` methods:

- `addConnectionPool(String poolName, JDBCLogin login, int minNumberOfConnections, int maxNumberOfConnections)`: creates a new connection pool and adds it to the pools managed by the `ServerSession`
- `useReadConnectionPool(int minNumberOfConnections, int maxNumberOfConnections)`: configures the read connection pool
- `useExclusiveReadConnectionPool(int minNumberOfConnections, int maxNumberOfConnections)`: configures the read connection pool to allow only a single thread to access each connection
- `setMaxNumberOfNonPooledConnections(int maxNumber)`: sets the maximum number of non-pooled connections

Connection options

TopLink provides a connection policy object that allows the application to customize the way connections are used within a server session object.

ClientSession connection options

There are four ways of getting connections from within a `ClientSession` object (these correspond to the four `acquireClientSession()` methods on the `ServerSession`):

- Acquire a `ClientSession` using the zero argument version of `acquireClientSession()`. This makes use of the default connection pool.
- Acquire a `ClientSession` by passing in a `poolName` as an argument to `acquireClientSession()`. This returns a `ClientSession` that uses a connection from the pool, `poolName`.
- Acquire a `ClientSession` by passing a `DatabaseLogin` object as an argument to `acquireClientSession()`. This returns a `ClientSession` that uses the `DatabaseLogin` object to obtain a connection.

These methods use a lazy database connection by default. This means that the connection is not allocated until a `UnitOfWork` is committed to the database. If you do not want to use a lazy database connection, but instead require that the database connection be established immediately, you must acquire a `ClientSession` by

passing a `ConnectionPolicy` object as an argument to `acquireClientSession()`. This allows you to use any of the three connection options (by setting up the `ConnectionPolicy` object properly) but also allows you to specify a lazy connection.

Connection policies

The `ConnectionPolicy` class provides the following methods for configuring a client connection:

- `setPoolName(String poolName)`: Sets up a connection from the connection pool. Alternatively, this can also be accomplished using `ConnectionPolicy(String poolName)`
- `setLogin(DatabaseLogin login)`: Sets up a connection by loggin directly into the database. Alternatively, this can also be accomplished using `ConnectionPolicy(DatabaseLogin login)` of the connection policy constructor.
- `useLazyConnection()`: specifies a lazy connection
- `setLazyConnection(boolean isLazy)`: specifies a lazy connection
- `dontUseLazyConnection()`: specifies an active connection
- If no database connections are available when a request for one is made, the application waits for one to become available.

Reference

Table 2–2 and Table 2–3 summarize the most common public methods for `client` and `server session`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for `client` and `server session`, see the TopLink JavaDocs.

Table 2–2 Elements for ClientSession

Element	Default	Method Names
Executing a query object *		`executeQuery(DatabaseQuery query)`
Reading from the database *	not applicable	`readAllObjects(Class domainClass, Expression expression)` `readObject(Class domainClass, Expression expression)`
Release *		`release()`
Unit of work *	not applicable	`acquireUnitOfWork()`

* Required property

Table 2–3 Elements for ServerSession

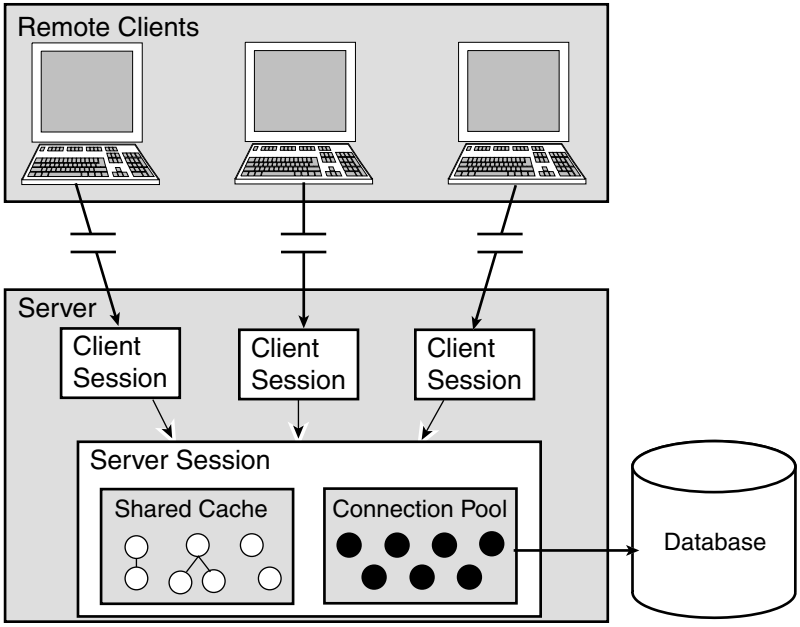
Element	Default	Method Names
Acquire Client Sessions	not applicable	`acquireClientSession()`
Logging	no logging	`logMessages()`
Login / logout	not applicable	`login()` `logout()`

Remote sessions

A remote session is a session that unlike other sessions actually resides on the client and talks to a server session on the server. Remote sessions handle object identity, proxies and the communication between the client and server layer.

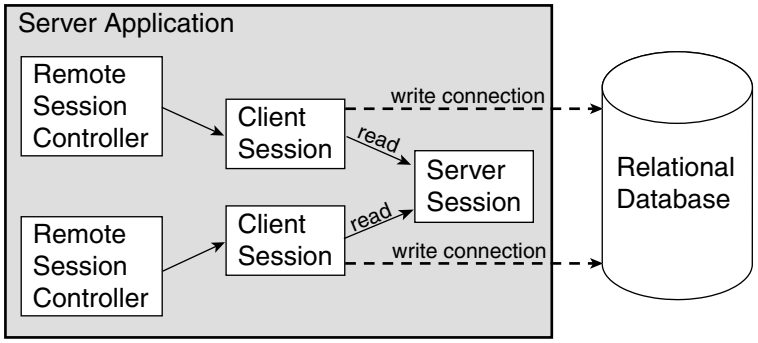
Figure 2–4 shows the TopLink client/server split. Much of the application logic runs on the client. The middle, dotted layer is implemented by TopLink and the application interacts with the remote session.

Figure 2–4 A model of a remote session for a three-tier application



The remote session can interact to a database session or a client session (see Figure 2–5). This set-up is done on the server side, by the user. Interaction between the remote session and the database session is not very useful in a distributed environment, because only a single user can interact with the database. However, if the remote session interacts with the client session, then multiple remote sessions can interact with the single database. The remote session can also reap the benefits of connection pooling.

Figure 2–5 The remote session and a database or a client session



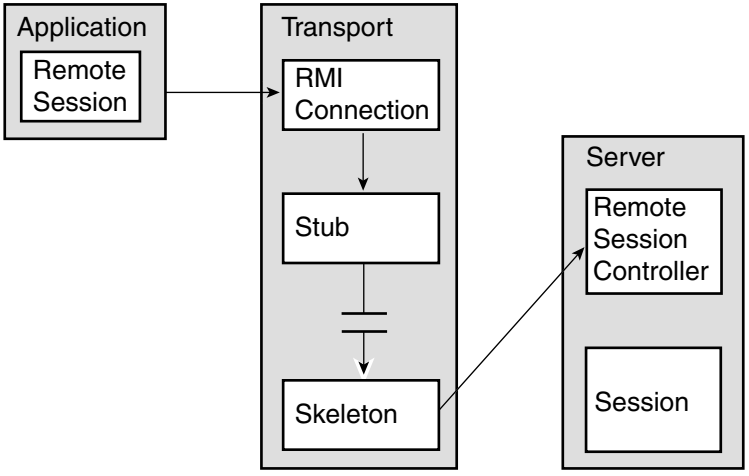
Architectural overview

The model consists of the following layers (see Figure 2–6):

- the application layer – client side application talking to remote session
- the transport layer – a session broker-specific layer
- the server layer – an application server setup talking to the TopLink server

The request from the client application to the server travels down through the layers of distributed system. A client making a request to the server session actually makes use of the remote session as a conduit to the server session. The client holds reference to a remote session. If necessary, the remote session forwards a request to the server session via the transport and server layer.

Figure 2–6 An architectural overview of the remote session



Application layer

The application layer consists of the application and the remote session. The remote session is a subclass of the session. The remote session handles all the public protocols of the session, giving the appearance of working with the local database session.

The remote session maintains its own identity map and a hash table of all the descriptors read from the server. If the remote session is able to handle a request by itself, the request is not passed to the server. For example, a request to read an object that has already been read is processed by the remote session. However, if the object is being read for the first time, the request is passed to the server session.

The remote session interacts to the transport layer through a remote connection.

Transport layer

The transport layer is responsible for carrying the semantics of the invocation. It is a broker-dependant layer that hides all of the broker-related dependencies from the application and server layer.

It consists of a remote connection that is an abstract entity. All the requests to the server are forwarded though the remote connection. Each remote session holds on to a single remote connection. The remote connection marshals and unmarshals all requests and responses on the client side.

In an RMI system, the remote connection interacts with an RMI stub/skeleton layer to talk to the server layer.

Remote session supports communicating over RMI and CORBA. It includes deployment classes and stubs for RMI, WebLogic RMI, VisiBroker, OrbixWeb, WebLogic EJB and Oracle 9i EJB.

Server layer

The server layer consists of a remote session controller dispatcher, a remote session controller, and a session. The remote session controller dispatcher marshals and unmarshals all responses and requests from the server side. This is a server side component.

The remote session controller dispatcher is an interface between the session and transport layers. It hides the broker-specific transport layer from the session.

Accessibility issues

The accessibility of the server running on a remote machine is a very sensitive issue because security of the server is very important. In such an environment, registering a remote session controller dispatcher as service can be detrimental as anyone can get access to the service and therefore to the entire database. The recommended set-up is to run some sort of server manager as a service that holds the remote controller session dispatcher. All the clients talk to the server manager and it implements the security model for accessing the remote session controller dispatcher.

On the client side, the user can get access to the server manager as it is a public service running on the server. Once the client gets access to the server manager, it can ask for the remote session controller dispatcher. The manager returns one if it qualifies the security model built into the server manager.

A remote connection is then created using the remote session controller dispatcher on the client side. Once the connection is created, the remote session is acquired from the remote connection. The API for the remote session is same as for the session. For the user, there is no difference between working on a session or a remote session.

The remote session maintains lots of processing behavior so as to minimize its interaction with the server session. It maintains an identity map to preserve the identity of an object. At runtime, the remote session builds its knowledge base by reading descriptors and mappings from the server side only when they are needed. These descriptors and mappings are light-weight because not all of the information

is passed on to the remote session. The information needed to traverse an object tree and to extract primary keys from the given object is passed with the mappings and descriptors.

Queries

Only read queries are publicly available on the client side. Object modification is done only through the unit of work.

Refreshing

Normal refreshing calls on the remote session force database hits and possible cache updates provided that the data were previously modified in the database. It could lead to poor performance and may refresh on queries when it is not desired; for example, the server session cache is positively known to be synchronized with the database.

Refresh operations against the server session cache are supported on the remote session. The descriptor can be configured to always remotely refresh the objects in the cache on all queries. This ensures that all queries against the remote session refresh the objects from the server session cache, without the database access.

Cache hits on remote sessions still occur on read object queries based on the primary keys. If these are not desired, the remote session cache hits on read object queries based on the primary key can be disabled.

Example 2–1 Remote session refreshes on the server session cache

```
// Remote session begin transaction
remoteSession.beginTransaction();

// Get the PolicyHolder descriptor
Descriptor holderDescriptor = remoteSession.getDescriptor(PolicyHolder.class);

// Set refresh on the server session cache
holderDescriptor.alwaysRefreshCachedOnRemote();

// Disable remote cache hits, ensure all queries go to the server session cache
holderDescriptor.disableCacheHitsOnRemote();
```

Indirection

Indirection objects are supported on the remote session. This is a special kind of value holder that can be invoked remotely on the client side. When invoked, the

value holder first checks to see if the requested object exists on the remote session. If not, then the associated value holder on the server is instantiated to get the value that is then passed back to the client. Remote value holders are used automatically; the application's code does not change.

Cursored streams

Cursored streams are supported remotely and are used in the same way as on the server.

Note: Scrollable cursors are not currently supported for remote sessions.

Unit of work

All object modifications must be done through the unit of work that is acquired from the remote session. For the user, this unit of work is the same as a normal unit of work acquired from the client session or the database session.

Creating a remote connection using RMIConnection

The goal of the following example is to create a remote TopLink session on a client that communicates with a remote session controller on a server using RMI. Once the connection has been created, the client application can use the remote session as it would any other TopLink session.

We will assume we have created an object on the server called `RMIServerManager` (not part of TopLink). This class has a method that instantiates and returns a `RMIRemoteSessionController` (a TopLink server side interface).

The following client-side code gets a reference to our `RMIServerManager` and then uses this to get the `RMIRemoteSessionController` running on the server. The reference to the session controller is then used in creating our `RMIConnection` from which we get a remote session.

Example 2–2 Client acquiring RMIRemoteSessionController from Server

```
RMIServerManager serverManager = null;
// Set the client security manager
try {
    System.setSecurityManager(new RMISecurityManager());
    } catch(Exception exception) {
    System.out.println("Security violation " + exception.toString());
```

```
}
// Get the remote factory object from the Registry
try {
    serverManager = (RMIServerManager) Naming.lookup("SERVER-MANAGER");
    } catch (Exception exception) {
    System.out.println("Lookup failed " + exception.toString());
}
// Start RMIRemoteSession on the server and create an RMIConnection
RMIConnection rmiConnection = null;
try {
    rmiConnection = new
    RMIConnection(serverManager.createRemoteSessionController());
    } catch (RemoteException exception) {
    System.out.println("Error in invocation " + exception.toString());
}
// Create a remote session which we can then use as a normal TopLink Session
Session session = rmiConnection.createRemoteSession();
```

The following code is used by `RMIServerManager` to create and return and instance of an `RMIRemoteSessionController` to the client. The controller sits between the remote client and the local TopLink session.

Example 2–3 Server creating RMIRemoteSessionController for Client

```
RMIRemoteSessionController controller = null;
try {
    // Create instance of RMIRemoteSessionControllerDispatcher which implements
    RMIRemoteSessionController. The constructor takes a TopLink session as a
    parameter.
    controller = new RMIRemoteSessionControllerDispatcher (localTOPLinkSession);
}
catch (RemoteException exception) {
    System.out.println("Error in invocation " + exception.toString());
}
return controller;
```

Session broker

The session broker is the mechanism provided by TopLink for multiple database access. Using the session broker, you can store the objects within an application on multiple databases.

The session broker:

- provides transparent multiple database access through a single TopLink session

- allows objects within an application to have references to objects stored on other databases
- allows for the application to have a single session view of multiple databases
- supports objects having references across databases
- transparently controls on which databases objects are stored
- manages single unit of work and transaction across multiple databases
- supports two-phase commit when integrated with a compliant JTS driver; otherwise, uses a two stage algorithm

Two-phase/two-stage commits

A two-*phase* commit is supported through integration with a compliant JTS driver (refer to the section "Java Transaction Service (JTS)" on page 2-28 for more details). A true two-phase commit is guaranteed to entirely pass or entirely fail even if a failure occurs during the commit.

If there is no integration with a JTS driver, the broker uses a *two-stage* commit algorithm. A two-*stage* commit differs slightly from a two-phase commit. The two-stage commit performed by the session broker is guaranteed except for failure during the final commit of the transaction, after the SQL statement has been successfully executed.

Using the session broker

After the session broker is set up and logged in, it is interacted with just like a session, making the multiple database access transparent. However, creating and configuring a `SessionBroker` is slightly more involved than creating a regular `DatabaseSession`.

Before using the `SessionBroker`, the sessions must be registered with it. To register a session with a `SessionBroker`, use the `registerSession(String name, Session session)` method. Before registration, all of the session's descriptors must have already been added to the session but not yet initialized. The sessions should not yet be logged in, as the session broker logs them in.

Example 2–4 Setting up two sessions with a session broker

```
Project p1 = ProjectReader.read(("C:\Test\Test1.project"));
Project p2 = ProjectReader.read(("C:\Test\Test2.project"));
```

```
//modify the user name and password if they are not correct in the
.project file
p1.getLogin().setUserName("User1");
p1.getLogin().setPassword("password1");
p2.getLogin().setUserName("User2");
p2.getLogin().setPassword("password2");
DatabaseSession session1 = p1.createDatabaseSession();
DatabaseSession session2 = p2.createDatabaseSession();

SessionBroker broker = new SessionBroker();
broker.registerSession("broker1", session1);
broker.registerSession("broker2", session2);

broker.login();
```

When the login method is performed on the session broker, both sessions are logged in and the descriptors in both sessions are initialized. After login, the session broker is treated like a regular session. TopLink handles the multiple database access transparently.

Example 2–5 Writing to the database using the session broker

```
UnitOfWork uow = broker.acquireUnitOfWork();
Test test = (Test) broker.readObject(Test.class);
Test testClone = uow.registerObject(test);
. . .
//change and manipulate the clone and any of its references
. . .
uow.commit();

//log out when finished
broker.logout();
```

Using the session broker in a three-tier architecture

Using the session broker in a three-tier architecture is very similar to the way it is used in two-tier. However, the client sessions must also be registered with a `SessionBroker`. The `ServerSessions` are set up in a similar way.

Example 2–6 Setting up the session broker in a three-tier architecture

```
Project p1 = ProjectReader.read(("C:\Test\Test1.project"))

Project p2 =
```

```
ProjectReader.read(("C:\Test\Test2.project"));

Server sSession1 = p1.createServerSession();
Server sSession2 = p2.createServerSession();

SessionBroker broker = new SessionBroker();
broker.registerSession("broker1", sSession1);
broker.registerSession("broker2", sSession2);
broker.login();
```

A client session can then be acquired from the server session broker, through the `acquireClientSessionBroker()` method.

Example 2–7 A sample client request code

```
Session clientBroker = broker.acquireClientSessionBroker();
return clientBroker;
```

Creating multiple projects in the Mapping Workbench

The session broker is designed to work with a project assigned to each session within the broker. There are a few ways to accomplish this in TopLink, but the following steps show the recommended approach.

1. Map your entire object model as you would normally in one single project.
2. Make a copy of the entire project, either by using the **Save as** menu option or by making a file copy.
3. In one of the projects, deactivate all the descriptors that do not reside on the database for which this project is being built.

   Note: Some of the items may show up as having errors; you can ignore these errors.

4. Repeat Step 3 for the other project. You should now have one project split into two.

   Note: This example assumes that only two projects are used however, the technique is identical for more than two projects.

Limitations

Using the session broker is not the same thing as linking databases at the database level. If your database allows linking, that is the recommended approach to providing multiple database access.

The session broker has the following limitations:

- Multiple table descriptor cannot be split across databases.
- Each class can live on only one database.
- Cannot use joins through expressions across databases.
- Many-to-many join tables and direct collection tables must be on the same database as the source object (<link>"Advanced use" describes a work-around using an amendment to the descriptor).

Advanced use

Many-to-many join tables and direct collection tables must be on the same database as the source object, because a join across both databases would be required on a read. However, it is possible to get around this by using the `setSessionName( String sessionName)` method on `ManyToManyMapping` and `DirectCollectionMapping`.

This method can be used to tell TopLink that the join table or direct collection table is on the same database as the target table.

```
Descriptor desc = session1.getDescriptor(Employee.class);
((ManyToManyMapping)desc.getObjectBuilder().getMappingForAttributeName("projects
")).setSessionName("broker2");
```

A similar method exists on `DatabaseQuery` that is used mostly for data queries (that is, non-object queries).

Reference

Table 2–4 summarizes the most common public methods for `SessionBroker`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for `SessionBroker`, see the TopLink JavaDocs.

Table 2–4 Elements for the SessionBroker

Element	Default	Method Names
Writing objects	not applicable	`acquireUnitOfWork()`
Acquiring client sessions	not applicable	`acquireClientSessionBroker()`
Database connection	not applicable	`login()` `logout()`

Java Transaction Service (JTS)

This section describes how TopLink for Java can be integrated with a transaction service satisfying the Java Transaction Service (JTS) API to participate in distributed transactions.

Review of transactions and transaction management

One of the important properties of databases is that transactions are *atomic*: a transaction either succeeds completely, or does not take effect at all. We get this automatically from most databases, but problems arise when we need to talk to more than one database at a time.

Consider the situation where we have bank accounts in two different databases. To transfer money from a checking account to a savings account, we want to withdraw money from an account in database A, and deposit it in an account in database B. We can use separate transactions for each database, but if a failure occurs on one database but not the other, then the balances will be incorrect. We need a single, unifying transaction that spans both databases.

Because updating information takes time and there is always a period during the transaction when the information is inconsistent, updating multiple databases may inevitably lead to situations where the information stored is inconsistent. A transaction can be described in more formal terms as a related set of operations with four properties. These are known by the acronym, "ACID":

Atomicity All operations are considered as a unit, that is either all the operations complete, leaving the information in its consistent amended state (known as committing), or all the operations are undone, leaving the information in its original consistent state (known as rollback).

Constancy The operations take the information held from one consistent state to another in a predictable fashion.

Isolation The partially updated states of the information are not visible outside the transaction itself.

Durability The outcome of the transaction is not reversed (partially or completely) after the transaction is completed.

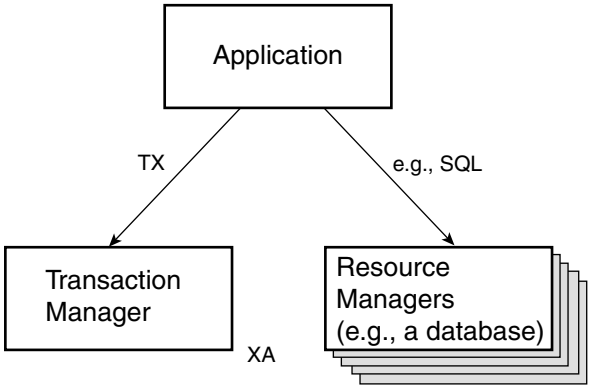
Distributed transactions

When we described the banking transaction in the "Review of transactions and transaction management" on page 2-28, it was assumed that all of the information necessary to complete the transaction was available locally. However, there are many valid business reasons why information must be stored on different machines. Information may be distributed according to geography. For example, the Sales database may be divided into 'Northern Region' and 'Southern Region'. The information may be divided along departmental lines, with the Accounting department holding billing information while the Stock department holds inventory details. Whatever the reasons for distributing the information, the business user still requires that all of the ACID properties of 'regular' transactions are also true of distributed transactions.

Transaction managers

In a non-distributed transaction, it is up to the single database to ensure the ACID properties of a transaction. In a distributed transaction there has to be careful co-operation between the various resources; thus The Open Group (formerly X/Open) has defined a formal model for Distributed Transaction Processing (DTP) known as the *three box model*. This model recognizes that there are three distinct components in a distributed transaction.

Figure 2–7 The Open Group DTP processing model for distributed transactions



The application implements the business logic and does not have direct access to a database. Rather, it interacts with resource managers via a programming interface, typically SQL for relational databases. In addition, the application interacts with a transaction manager to begin and end a transaction. This sets up the *transaction context* within which all the components operate.

The resource managers have direct access to information and other database-specific resources. Typically a resource manager *is* a database, but it can be anything that is capable of transactional work (for example, a secure printer). The interface to the resource manager does not reveal any transaction details; rather, the resource manager interacts with the transaction manager to determine the current *transaction context*.

The transaction manager is dedicated to coordinating the activities of all of the participants in the transaction. It provides the TX interface so that applications can initiate transactional work. It co-ordinates the resource managers via the XA interface. The prime responsibility of the transaction manager is to guide the two-phase commit process that allows outstanding changes held by all the resource managers to be properly written to backing-store.

Two-phase commit with presumed rollback

The *two-phase commit with presumed rollback* model (2-PC) allows resource managers to make temporary changes during the transaction so that they can be applied at the end of the transaction (committed) or undone (rolled back). During the transaction there is no ambiguity if a failure occurs -- all temporary changes are undone. When

the transaction is committed by the application, then the temporary changes are made permanent in two phases.

In phase one, each resource (represented by a resource manager) is told to *prepare*. At this stage it must store in a secure way the changes it is about to make together with a secure record of its action. If it fails to do this, then it must vote *rollback*. If it succeeds in securing its records, then it must vote *commit* and wait for the final decision of the transaction manager. Once a resource has voted to commit, it gives up the right to rollback.

When all resources have voted on the outcome of the transaction or a failure has occurred, the transaction manager decides the final outcome of the transaction.

- If any resource votes to rollback, or a failure occurs, then the transaction manager rolls the transaction back by informing each participant.
- If the transaction manager receives a commit vote from every participant, then it takes the commit decision and records this securely.
- When the commit or rollback decision is taken, the transaction reaches that conclusion whatever failures may occur.

In phase two, each resource is told to commit. The resources must then make their temporary changes permanent and forget the record of their action made when they voted to commit. Once they have forgotten the secure record of the transaction, they can report *done* to the transaction manager. When the transaction manager has received *done* from all participants, it can forget the secure record of the transaction in its turn, and the transaction is complete.

Relationship between OMG Object Transaction Service (OTS) and Java Transaction Service (JTS)

The OMG Object Transaction Service defines interfaces that allow multiple, distributed objects to provide and participate in distributed ACID transactions. It is upon this specification that the Java Transaction Service (JTS) is based.

- For more information about OTS, please consult the OMG documentation at www.omg.org/docs.
- For more information about JTS, please consult the Sun documentation at java.sun.com/products/jts.

JTS transaction synchronization

Transaction synchronization allows interested parties to get notification from the transaction manager about the progress of the commit. For each transaction started, an application may register a `javax.transaction.Synchronization` callback object that implements the following methods:

- `beforeCompletion` method is called prior to the start of the two-phase transaction complete process. This call is executed in the same *transaction context* of the caller who initiates the begin.
- `afterCompletion` method is called after the transaction has completed. The status of the transaction is supplied as the parameter. This method is executed **without** a *transaction context*.

The `Synchronization` interface described can be thought of as a lightweight 'listener' to the lifecycle of the global external transaction. It is through this interface that TopLink can participate in a global external transaction by registering a `Synchronization` callback object for a unit of work.

TopLink unit of work and the synchronization interface

The TopLink `Session` must be configured with an instance of a class that implements the `ExternalTransactionController` interface (from package `oracle.toplink.sessions`).

TopLink includes an external transaction controller for JTS 0.95. This controller is also compatible with JTS and JTA up to and including the JTA 1.0.1 specification. The controllers included with TopLink are found in the `oracle.toplink.jts` and oracle.toplink`.jts.wls` packages. These packages include generic JTS Listener and Manager classes, as well as classes that specifically support a number of databases and application servers including

- Inprise
- Oracle9i
- OS390
- BEA WebLogic Server
- IBM WebSphere Server

If your JTS driver is not compatible with these versions you can build your own implementor of the `ExternalTransactionController` interface.

When using the JTS transaction controller, the transaction manager must be set in the `JTSSynchronizationListener` class. The transaction manager is required to give TopLink access to the global JTS transaction. Unfortunately there is no standard way to access the transaction manager so you must consult your JTS driver documentation to determine how to access this. When using the WebLogic JTS controller this is not required.

Example 2–8 Configuring ExternalTransactionController on the TopLink session

```
... (appropriate import stmts)
Project project = Project.read("C:\myDir\myProj.project");
// login specifics (database URL, etc) comes from the project
DatabaseLogin login = project.getLogin();
/* set External behaviours: connectionPooling,
Transaction mgmt, Transaction controller.
Must be done before Session is created
*/
login.useExternalTransactionController();
login.useExternalConnectionPooling();
ServerSession session = project.createServerSession();
// The transaction manager must be set
JTSSynchronizationListener.setTransactionManagerjtsTransactionManager);
session.setExternalTransactionController(new
JTSExternalTransactionController());
...
```

Writing to a database in three-tier environment

Use a Unit of Work to write to a database that uses JTS externally-controlled transactions. To do this successfully, however, you must ensure that there is only one unit of work associated with a given transaction. To do so, check for a transaction and associated unit of work as follows:

```
UnitOfWork uow = serverSession.getActiveUnitOfWork();
```

The following logic is executed:

- If there is no current external transaction in progress return null;
- If there is a current external transaction and it has an associated unit of work, return this unit of work;
- If there is a current external transaction and it has no associated unit of work associated with current external transaction
  - create a client session

- acquire a new unit of work
- returns this unit of work

> **Note:** While there are other ways to write to a database through a JTS external controller, using the method described here ensures that all units of work are completed successfully.

External connection pools and external transaction control

From the example on the previous page, we can see that in addition to providing an `ExternalTransactionController` for the `Session`, the `DatabaseLogin` needs two additional properties configured:

- `useExternalTransactionController()` - To interact correctly with a JTS service, we need to indicate to the `DatabaseLogin` object that transaction control is being managed by an external entity.
- `useExternalConnectionPooling()` - It is common among JTS implementations that access to the service is 'wrapped' and presented as a 'regular' JDBC driver. For example, the WebLogic JTS service is available as "`weblogic.jdbc.jts.Driver`". This driver (and its corresponding connection string "`jdbc:weblogic:jts:{a_pool_name}`" implements a *pool* of JDBC connections that can be configured separately from the login information (please consult the WebLogic product documentation for more information).

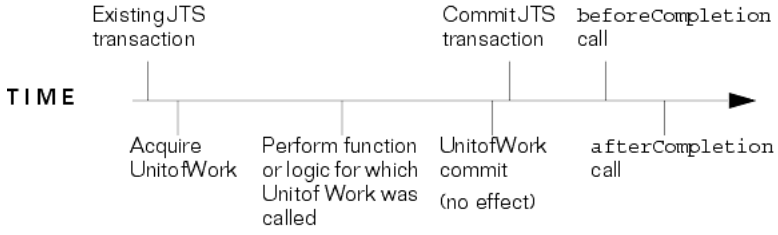
> **Note:** For some JTS implementations, it is not necessary to configure all three properties. If you are using a JTS service from a vendor not mentioned here, please contact Technical Support for information on the appropriate configuration settings for your particular JTS implementation.

A user acquires a `UnitOfWork` from the TopLink session using the standard API `acquireUnitOfWork()`. Within `acquireUnitOfWork()`, registration of a `Synchronization` object with the current transaction is delegated to the ETC. If no global external transaction exists, the unit of work begins its own JTS transaction. In this case, if the unit of work is committed it also commits the JTS transaction that it began.

The user manipulates the `UnitOfWork` in the usual fashion, registering objects and altering clone copies (see "Using units of work" on page 1-44). At this point, there are two scenarios to consider.

Scenario 1 The user calls `uow.commit()` before the completion of the global external transaction – that is, neither `Synchronization` callbacks has yet occurred (see Figure 2–8).

Figure 2–8 External transaction exists when UnitOfWork is called

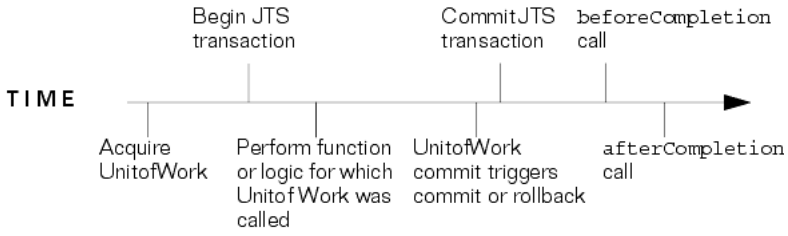


At `uow.commit()` time, a flag is set in the `UnitOfWork` indicating that a merge is pending. In the `beforeCompletion` callback, the appropriate SQL is sent to the database; if during this operation an `OptimisticLockException` (or some other `RuntimeException`) is thrown, the `UnitOfWork` is marked 'dead' and the global external transaction is rolled back using the standard JTS APIs.

If the `afterCompletion` callback indicates success, the clones are merged with the TopLink `Session`. If the `afterCompletion` callback indicates failure (and possibly the `beforeCompletion` callback is not even invoked), the merge is **not** done and the `UnitOfWork` is released.

Scenario 1 No global external transaction exists when the user acquires a unit of work (see Figure 2–9).

Figure 2–9 No external transaction exists when UnitOfWork is called



In this case, the `beforeCompletion` callback or the `afterCompletion` callback causes the unit of work to commit and if successful the `afterCompletion` callback causes the unit of work to merge its changes into the session cache. If the JTS transaction fails or is rolled back, the unit of work is released.

Table 2–5 Public API for JTS

Class or Interface	API
`oracle.toplink.sessions.DatabaseSession (I)`	
	`ExternalTransactionController getExternalTransactionController()` `setExternalTransactionController( ExternalTransactionController etc)`
`oracle.toplink.sessions.DatabaseLogin (C)`	

	`dontUseExternalConnectionPooling()`
	`useExternalConnectionPooling()`
	`usesExternalConnectionPooling()`
	`dontUseExternalTransactionController()`
	`useExternalTransactionController()`
	`usesExternalTransactionController()`

Extending TopLink's JTS capabilities

Since the JTS specification is new, vendors have implemented their JTS service against a changing backdrop, the JTS specification itself. To accommodate this, TopLink's JTS integration implementation is flexible to allow for local modifications.

A example of an implementation of a JTS External Transaction Controller is found in the package oracle.toplink.`jts`. Unfortunately, there needs to be different concrete implementations of the `AbstractSynchronizationListener` interface

because the JTS specification has been changing recently. A vendor-specific implementation suitable for BEA WebLogic's JTS implementation is found in the package `oracle.toplink.jts.wls`.

In the package `oracle.toplink.jts.`, two abstract classes form the basis of any local modifications:

Table 2–6 Public API for JTS local modifications

Class or Interface	API
`oracle.toplink.jts.AbstractExternalTransactionController (A)`	
	`register(UnitOfWork uow, Session session) throws Exception`
`oracle.toplink.jts.AbstractSynchronizationListener (A)`	
	`rollbackGlobalTransaction()` `boolean wasTransactionCommitted( int status)`

Extensions to TopLink's JTS capabilities thus are always a *pair* of concrete classes that extend the named classes. A subclass of `AbstractExternalTransactionController` must implement the abstract methods from Table 2–6.

The `register` method performs a simple function – it delegates the call; it must invoke the static `register` method on the specific subclass of `AbstractSynchronizationListener` that is 'paired' with the controller class – for example, the `JTSExternalTransactionController` implements register as follows:

```
public void register(UnitOfWork uow, SynchronizationListener sl, Session
session) throws Exception {
    JTSSynchronizationListener.register(uow, sl, session);
}
```

A subclass of `AbstractSynchronizationListener` must implement the two abstract methods from Table 2–6 as well as the static `register` method mentioned above.

Abstract methods of AbstractSynchronizationListener requiring concrete implementation for local JTS modifications

```
/** This method must be re-written for the concrete implementations of
XXXSynchronizationListener as the various revisions of JTS that vendors have
```

```
written their JTS implementations against have different ways of referring
to/dealing with the 'Transaction' object
*/
public abstract void rollbackGlobalTransaction();
/** Examine the status flag to see if the Transaction committed. This method
must be re-written for the concrete implementations of
XXXSynchronizationListener as the various revisions of JTS that vendors have
written their JTS implementations against have different status codes
 */
public abstract boolean wasTransactionCommited(int status);
```

For example, the `JTSSynchronizationListener` implements `register` as follows:

Prototypical implementation of register for JTS service

```
...
import javax.transaction.*;
...
public static void register(UnitOfWork uow, SynchronizationListener sl, Session
session) throws Exception {
    Transaction tx = tm.getTransaction();
    JTSSynchronizationListener jsl = new JTSSynchronizationListener(uow,
    sl,session,tx);
    tx.registerSynchronization(jsl);
}
```

In the previous example implementation, the current global transaction is acquired from `tm`, a static variable local to `JTSSynchronizationListener` that must be set to an instance of a class that implements the `javax.transaction.TransactionManager` interface.

For the abstract methods, the `JTSSynchronizationListener` implements `rollbackGlobalTx` and `txCommited` as follows:

Example 2–9 Example implementation of rollbackGlobalTransaction and wasTransactionCommited for JTS service

```
public void rollbackGlobalTransaction() {
    try {
        ((Transaction) globalTx).setRollbackOnly();
    }
    catch (SystemException se) {
    }
}
public boolean wasTransactionCommited(int status) {
```

```
    if (status == Status.STATUS_COMMITTED)return true;
    else return false;
}
```

To contrast, the `WebLogicJTSSynchronization` implements these methods as follows:

Example 2–10 Concrete implementation of `register` for WebLogic's JTS service

```
...import weblogic.jts.common.*;
import weblogic.jts.internal.*;
import weblogic.jndi.*;
import javax.naming.*;
import java.util.*;
import java.io.*;
...
public static void register(UnitOfWork uow, Session session) throws
    Exception {
    Context ctx = null;
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, WEBLOGIC_FACTORY);
    env.put(Context.PROVIDER_URL,providerUrl);
    // these statics are null by default; check if someone set them
    if (principal != null)env.put(Context.SECURITY_PRINCIPAL,
    principal);
    if (credentials != null) env.put(Context.SECURITY_CREDENTIALS,credentials);
    if (authentication != null)env.put(Context.SECURITY_AUTHENTICATION,
    authentication);
    ctx = new InitialContext(env);
    Current current = (Current)ctx.lookup("javax.jts.UserTransaction");
    WebLogicJTSSynchronization wjs = new
    WebLogicJTSSynchronization(uow,session,current);
    current.getControl().getCoordinator()
    .registerSynchronization(wjs);
    }
public void rollbackGlobalTransaction() {
    ((Current) globalTx).setRollbackOnly();
}
    public boolean wasTransactionCommited(int status) {
    if (status == Synchronization.COMPLETION_COMMITTED)
    return true;
    else
    return false;
}
```

TopLink support for Java Data Objects (JDO)

TopLink provides an enterprise-proven architecture for the persistence of Java objects and JavaBeans to relational databases, object-relational databases and enterprise information systems. The TopLink architecture and API have evolved through over a decade of development and usage across many vertical markets, countries and applications. Included in this persistence architecture is support for Java Data Objects (JDO).

JDO is an API for transparent database access. The JDO architecture defines a standard API for data contained in local storage systems and heterogeneous enterprise information systems, such as ERP, mainframe transaction processing, and database systems. JDO enables programmers to create code in Java that transparently accesses the underlying data store without using database-specific code.

TopLink provides basic JDO support based on JDO Proposed final draft 1.0 specification (for information on the specification, see the Sun Microsystems web site at java.sun.com.

TopLink's support for JDO includes much of the JDO API, but does not require the class to be enhanced or modified by JDO Reference Enhancer aspects of the JDO specification and other JDO products.

Caution: JDO is a persistence specification that is in the proposal stage. As such, it may undergo major changes in future editions, or even be abandoned altogether. As a result, a thorough and proper evaluation of JDO compared to other architectures supported by TopLink is strongly recommended before putting a JDO-based architecture into production.

Understanding the JDO API

The JDO API consists of four main interfaces:

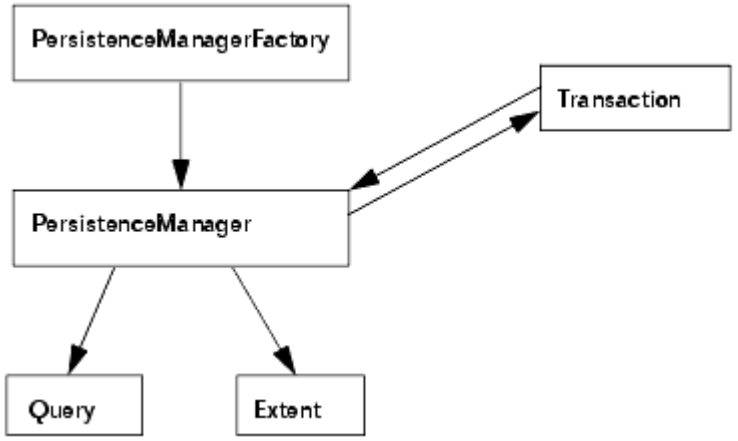
PersistenceManagerFactory A factory that generates PersistenceManagers. It has configuration and login API.

PersistenceManager The main point of contact from the application. It provides API for accessing the transaction, queries and object life cycle API (`makePersistent`, `makeTransactional`, `deletePersistent`).

Transaction Defines basic begin, commit, rollback API.

Query Defines API for configuring the query (filter, ordering, parameters, and variables) and for executing the query.

Figure 2–10 Understanding the JDO API



JDO implementation

TopLink implements the main JDO interfaces `PersistenceManagerFactory`, `PersistenceManager`, and `Transaction`. It extends the query functionality to include the complete TopLink query framework. The supported APIs are listed in the reference tables of their respective implementation class. JDO APIs that are not listed in the reference tables are not supported.

For more information on the TopLink query framework, see "Using the query framework" on page 1-22.

JDOPersistenceManagerFactory

The `JDOPersistenceManagerFactory` class implements a JDOPersistenceManagerFactory. This factory creates PersistenceManagers.

To create a JDOPersistenceManagerFactory, the constructor takes a session name string or a TopLink session or project. If the factory is constructed from a project, a new DatabaseSession is created and attached to the `PersistenceManager` every time it is obtained through the `getPersistenceManager` method.

Note: `DatabaseSession` is typically used for single-threaded applications. `ServerSession` should be used for multi-threaded application.

The `PersistenceManager` is not multi-threaded. For multi-threaded application, each thread should have its own PersistenceManager. The `JDOPersistenceManagerFactory` should be constructed from a ServerSession not DatabaseSession or Project to make use of the lighter weight client session and more scalable connection pooling.

Creating a JDOPersistenceManagerFactory The following code creates a factory from a TopLink session named "jdoSession" that is managed by SessionManager. The SessionManager manages a singleton instance of TopLink ServerSession or DatabaseSession named "jdoSession". Refer to SessionManager documentation for more info.

```
JDOPersistenceManagerFactory factory= new
JDOPersistenceManagerFactory("jdoSession");
//Create a persistence manager factory from an instance of
TopLink ServerSession or DatabaseSession that is managed by
the user.//

ServerSession session = (ServerSession) project.createServerSession();
JDOPersistenceManagerFactory factory= new JDOPersistenceManagerFactory(session);
//Create a persistence manager factory with ties to a
DatabaseSession that is created from TopLink project.//

JDOPersistenceManagerFactory factory= new JDOPersistenceManagerFactory(new
EmployeeProject());
```

Obtaining PersistenceManager New PersistenceManagers are created by calling the `getPersistentManager` method. If the factory is constructed from a Project instance, it can also configure the userid and password using `getPersistentManager(String userid, String password)`.

Reference Table 2–7 summarizes the most common public methods for `PersistenceManagerFactory`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for `PersistenceManagerFactory`, see the TopLink JavaDocs.

Table 2–7 Elements for PersistenceManagerFactory

Element	Default	Method Names
Construct a factory from a session named "default" that is managed by SessionManager		`JDOPersistenceManagerFactory()`
Construct a factory from a session name that is managed by SessionManager		`JDOPersistenceManagerFactory(String sessionName)`
Construct a factory from a user session		`JDOPersistenceManagerFactory(Session session)`
Construct a factory from a project		`JDOPersistenceManagerFactory(Project project)`
Query mode that specifies whether cached instances are considered when evaluating the filter expression	false	`getIgnoreCache()setIgnoreCache(boolean ignoreCache)`
Transaction mode that allows instances to be read outside a transaction	true	`getNontransactionalRead()` `setNontransactionalRead(boolean nontransactionalRead)`
These settings are enable only if the factory is constructed from a TopLink Project	user name, password, url, driver from project login	`getConnectionUserName()setConnectionUserName(String userName)getConnectionPassword()` `setConnectionPassword(String password)getConnectionURL()` `setConnectionURL(String URL)getConnectionDriverName()` `setConnectionDriverName(String driverName)`
Access PersistenceManager. The user id and password are set only if the factory is constructed from a TopLink Project. Otherwise, use default values.	User id, password from session login or project login	`getPersistenceManager()` `getPersistenceManager(String userid, String password)`
Non-configurable properties		`getProperties()`
Collection of supported option String		`supportedOptions()`

JDOPersistenceManager

The `JDOPersistenceManager` class implements a JDOPersistenceManager, the primary interface for JDO-aware application components. The JDOPersistenceManager is the factory for the Query interface and contains methods for accessing transactions and managing the persistent life cycle instances. The JDOPersistenceManager instance can be obtained from JDOPersistenceManagerFactory.

Inserting JDO objects New JDO objects are made persistent using the `makePersistent()` or `makePersistentAll()` methods. If the user does not manually begin the transaction, TopLink will begin and commit the transaction when either `makePersistent()` or `makePersistentAll()` is invoked. Note that if the object is already persisted, calling these methods has no effect.

Example 2–11 Persist a new employee named Bob Smith

```
Server serverSession = new EmployeeProject().createServerSession();
PersistenceManagerFactory factory = new
JDOPersistenceManagerFactory(serverSession);
PersistenceManager manager = factory.getPersistenceManager();
Employee employee = new Employee();
employee.setFirstName("Bob");
employee.setLastName("Smith");
manager.makePersistent(employee);
```

Updating JDO Objects JDO objects are modified using a transactional instance. The object is modified within a transaction context by manually beginning and committing the transaction.

A transactional object is an object that is subject to the transaction boundary. Transactional objects can be obtained several ways, including

- using `getObjectById()`
- executing a transactional-read query
- Using the TopLink extended API `getTransactionalObject()`

The transactional-read query is a query that is executed when the `nontransactionalRead` flag of the current transaction is false. The current transaction is obtained from the PersistenceManager by calling `currentTransaction()`.

Example 2–12 Update an employee

The following example illustrates how to add a new phone number to an employee object, modify its address and increase its salary by 10%.

```
Transaction transaction = manager.currentTransaction();
if(!transaction.isActive()) {
transaction.begin();
}
// Get the transactional instance of the employee//
Object id = manager.getTransactionalObjectId(employee);
Employee transactionalEmployee = manager.getObjectById(id, false);
transactionalEmployee.getAddress().setCity("Ottawa");
transactionalEmployee.setSalary((int) (employee.getSalary() * 1.1));
transactionalEmployee.addPhoneNumber(new PhoneNumber("fax", "613", "3213452"));

transaction.commit();
```

Deleting Persistent Objects JDO objects are deleted using either `deletePersistent()` or `deletePersistentAll()`. The objects can be transactional or non-transactional. If the user does not manually begin the transaction, TopLink will begin and commit the transaction when `deletePersistent ()` or `deletePersistentAll ()` is invoked.

It is important to understand that deleting objects using `deletePersistent()` or `deletePersistentAll()` is similar to deleting objects using UnitOfWork. When an object is deleted, its privately-owned parts are also deleted, because privately-owned parts cannot exist without their owner. At commit time, SQL is generated to delete the objects, taking database constraints into account. If an object is deleted, then the object model must take the deletion of that object into account. References to the object being deleted must be set to null or removed from the collection. Modifying references to the object is done through its transactional instance.

Example 2–13 Deleting a team leader from a project

```
Transaction transaction = manager.currentTransaction();
if(!transaction.isActive()) {
transaction.begin();
}
Object id = manager.getTransactionalObjectId(projectNumber);
Project transactionalProject = (Project) manager.getObjectById(id);
Employee transactionalEmployee = transactionalProject.getTeamLeader();
// Remove team leader from the project//
transactionalProject.setTeamLeader(null);
```

```
// Remove owner that is the team leader from phone numbers//
for(Enumeration enum = transactionalEmployee.getPhoneNumbers().elements();
enum.hasMoreElements();) {
    ((PhoneNumber) enum.nextElement()).setOwner(null);
}
manager.deletePersistent(transactionalEmployee);
transaction.commit();
```

Example 2–14 Deleting a Phone Number

```
Transaction transaction = manager.currentTransaction();
if(!transaction.isActive()) {
transaction.begin();
}
Object id = manager.getTransactionalObjectId(phoneNumber);
PhoneNumber transactionalPhoneNumber = (PhoneNumber) manager.getObjectById(id);
transactionalPhoneNumber.getOwner().getPhoneNumbers().remove(transactionalPhoneN
umber);
manager.deletePersistent(phoneNumber);
transaction.commit();
```

Obtaining Query TopLink does not support the JDO Query language but instead includes support within JDO for the more advanced TopLink query framework (for information on the TopLink query framework, see "Using the query framework" on page 1-22). A key difference is that, while the JDO query language requires results to be returned as a collection of candidate JDO instances (either a java.util.Collection,or anExtent, the result type returned by the TopLink query framework depends on the type of query used. For example, if a ReadAllQuery is used, the result is a Vector.

The query factory is supported through the following APIs.

- Standard API:

  ```
  newQuery();
  newQuery(Class persistentClass);
  ```

- TopLink extended API:

  ```
  newQuery(Class persistentClass, Expression expressionFilter);
  ```

> **Note:** Obtaining Query from a different newQuery() API could result in JDOUserException or getting the query implicitly created from the supported API.

A ReadAllQuery is created with the Query instance by default.

Reference Table 2–8 and Table 2–9 summarize the most common public methods for the Query API and TopLink extended API:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for the Query API and TopLink extended API, see the TopLink JavaDocs.

Table 2–8 Elements for Query API

Element	Default	Method Name
Release resource to allow garbage collection		`close()`
Transaction		`currentTransaction()`
Delete objects		`deletePersistent(Object object)deletePersistentAll(Collection objects)deletePersistentAll(java.lang.Object [] objects)`
Mark objects as no longer needed in the cache		`evict(Object object)evictAll()evictAll(Collection objects)evictAll(Object[] objects)`
Extent		`getExtent(Class queryClass, boolean readSubclasses)`
Cache mode for queries	Ignore cache from the persistence manager factory	`getIgnoreCache()setIgnoreCache(boolean ignoreCache)`
Obtain transactional state of object		`getObjectById(Object object, boolean validate)getTransactionalObjectId (Object object)`
A PersistenceManager instance can be used until it is closed		`isClosed()`

Table 2–8 Elements for Query API (Cont.)

Element	Default	Method Name
Insert objects		`makePersistent(Object object)makePersistentAll(Collection objects)makePersistentAll (Object[] objects)`
Make objects subject to transactional boundaries by registering them to UnitOfWork		`makeTransactional(Object object)makeTransactionalAll(Collection objects)makeTransactionalAll (Object[] objects)`
Query factory		`newQuery()newQuery(Class queryClass)`
Refreshing objects		`refresh(Object object)refreshAll()refreshAll (Collection objects)refreshAll (Object[] objects)`

Table 2–9 Elements for TopLink extended API

Element	Default	Method Name
Obtain transactional object		`getTransactionalObject(Object object)`
Query factory		`newQuery(Class queryClass, Expression expression)`
Reading objects		`readAllObjects(Class domainClass)readAllObjects(Class domainClass)readObject(Class domainClass, Expression expression)`

JDOQuery

The `JDOQuery` class implements the JDO Query interface. It defines API for configuring the query (filter, ordering, parameters, and variables) and for executing the query. TopLink extends the query functionality to include the full TopLink query framework (for information on the TopLink query framework, see "Using the query framework" on page 1-22). Users can customize the query to use advanced features such as batch reading, stored procedure calls, partial object reading, query by example, and so on. TopLink currently does not support the JDO query language, but users can use either SQL or EJBQL in the JDO Query interface. For more information on EJBQL support, see Chapter 4, "EJBQL Support".

Each JDOQuery instance is associated with a TopLink query. When the JDO Query is obtained from the PersistenceManager by calling a supported `newQuery` method, a new ReadAllQuery is created and associated with the query. JDO Query can reset its TopLink query to a specific type by calling `asReadObjectQuery()`, `asReadAllQuery()`, or `asReportQuery`.

Customizing the query using the TopLink Query Framework Much of the TopLink query framework functionality is provided through the public API. In addition, users can build complex functionality into their queries by customizing their own query. Users can create customized a TopLink query and associate it with the JDO Query by calling `setQuery()`.

Using a customized TopLink query gives users the complete functionality of TopLink query framework. An example for using customize query is using a DirectReadQuery with custom SQL to read the id column of the employee.

> **Note:** TopLink extended APIs are configured for a specific TopLink query type. Exception could be thrown if methods are used with the wrong query type. See Table 2–10 for correct usage.

Example 2–15 Use a ReadAllQuery to read all employees who live in New York

```
Expression expression = new
ExpressionBuilder().get("address").get("city").equal("New York");
Query query = manager.newQuery(Employee.class, expression);
Vector employees = (Vector) query.execute();
```

Example 2–16 Use a ReadObjectQuery to read the employee named Bob Smith

```
Expression exp1 = new ExpressionBuilder().get("firstName").equal("Bob");
Expression exp2 = new ExpressionBuilder().get("lastName").equal("Smith ");
JDOQuery jdoQuery = (JDOQuery) manager.newQuery(Employee.class);
jdoQuery.asReadObjectQuery();
jdoQuery.setFilter(exp1.and(exp2));
Employee employee = (Employee) jdoQuery.execute();
```

Example 2–17 Use a ReportQuery to report employee's salary

```
JDOQuery jdoQuery = (JDOQuery) manager.newQuery(Employee.class);
jdoQuery.asReportQuery();
jdoQuery.addCount();
jdoQuery.addMinimum("min_salary ",
jdoQuery.getExpressionBuilder().get("salary"));
```

```
jdoQuery.addMaximum("max_salary",
jdoQuery.getExpressionBuilder().get("salary"));
jdoQuery.addAverage("average_salary",
jdoQuery.getExpressionBuilder().get("salary"));
// Return a vector of one DatabaseRow that contains reported info
Vector reportQueryResults = (Vector) jdoQuery.execute();
```

Example 2–18 Use a customized DirectReadQuery to read employee 's id column.

```
DirectReadQuery TopLinkQuery = new DirectReadQuery();
topLinkQuery.setSQLString("SELECT EMP_ID FROM EMPLOYEE");
JDOQuery jdoQuery = (JDOQuery) manager.newQuery();
jdoQuery.setQuery(topLinkQuery);
// Return a Vector of DatabaseRows that contain ids
Vector ids = (Vector)jdoQuery.execute(query);
```

Reference Table 2–10 and Table 2–11 summarize the most common public methods for the JDO Query API and TopLink extended API:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for the JDO Query API and TopLink extended API, see the TopLink JavaDocs.

Table 2–10 Elements for JDO Query API

Element	Default	Method Name
Close Cursor result		`close(Object queryResult)`
Declare query parameters		`declareParameters(String parameters)`
Execute query		`execute()execute(Object arg1)execute(Object arg1, Object arg2)execute(Object arg1, Object arg2, Object arg3)executeWithArray(java.lang.Object[] arg1)executeWithMap(Map arg1)`
Cache mode for query result	Ignore cache from the persistence manager	`getIgnoreCache()setIgnoreCache(boolean ignoreCache)`
PersistenceManager		`getPersistenceManager()`

Table 2–10 Elements for JDO Query API (Cont.)

Element	Default	Method Name
ReadObjectQuery, ReadAllQuery, ReportQuery		`setClass(Class queryClass)`
ReadAllQuery		`setOrdering(String ordering)`

Table 2–11 Elements for TopLink Extended API

Element	Default	Method Name
Convert query		`asReadAllQuery()asReadObjectQuery()` `asReportQuery()`
Access TopLink query	ReadAllQuery	`getQuery()setQuery(DatabaseQuery newQuery)`

ReadObjectQuery, ReadAllQuery, ReportQuery		`acquireLocks()`
		`acquireLocksWithoutWaiting()`
		`addJoinedAttribute(String attributeName)`
		`addJoinedAttribute(Expression attributeExpression)`
		`addPartialAttribute(String attributeName)`
		`addPartialAttribute(Expression attributeExpression)`
		`checkCacheOnly()`
		`dontAcquireLocks()`
		`dontRefreshIdentityMapResult()`
		`dontRefreshRemoteIdentityMapResult()`
		`getExampleObject()`
		`getExpressionBuilder()`
		`setQueryByExampleFilter(Object exampleObject)`
		`setQueryByExamplePolicy(QueryByExamplePolicy newPolicy)`
		`setShouldRefreshIdentityMapResult (boolean shouldRefreshIdentityMapResult)`
		`shouldRefreshIdentityMapResult()`

ReadObjectQuery		`checkCacheByExactPrimaryKey()`
		`checkCacheByPrimaryKey()`
		`checkCacheThenDatabase()`
		`conformResultsInUnitOfWork()`
		`getReadObjectQuery()`

Table 2–11 Elements for TopLink Extended API (Cont.)

Element	Default	Method Name

ReadAllQuery		`addAscendingOrdering(String queryKeyName)`
		`addDescendingOrdering(String queryKeyName)`
		`addOrdering(Expression orderingExpression)`
		`addBatchReadAttribute(String attributeName)`
		`addBatchReadAttribute(Expression attributeExpression)`
		`addStandardDeviation(String itemName)`
		`addStandardDeviation(String itemName, Expression attributeExpression)`
		`addSum(String itemName)`
		`addSum(String itemName, Expression attributeExpression)`
		`addVariance(String itemName)`
		`addVariance(String itemName, Expression attributeExpression)`
		`getReadAllQuery()`
		`useCollectionClass(Class concreteClass)`
		`useCursoredStream()`
		`useCursoredStream(int initialReadSize, int pageSize)`
		`useCursoredStream(int initialReadSize, int pageSize, ValueReadQuery sizeQuery)`
		`useDistinct()useMapClass(Class concreteClass, String methodName)`
		`useScrollableCursor()`
		`useScrollableCursor(int pageSize)`

Table 2–11 Elements for TopLink Extended API (Cont.)

Element	Default	Method Name

		`addAttribute(String itemName)`
		`addAttribute(String itemName, Expression attributeExpression)`
		`addAverage(String itemName)`
		`addAverage(String itemName, Expression attributeExpression)`
		`addCount()`
		`addCount(String itemName)`
		`addCount(String itemName, Expression attributeExpression)`
		`addGrouping(String attributeName)`
		`addGrouping(Expression expression)`
		`addItem(String itemName, Expression attributeExpression)`
		`addMaximum(String itemName)`
		`addMaximum(String itemName, Expression attributeExpression)`
		`addMinimum(String itemName)`
		`addMinimum(String itemName, Expression attributeExpression)`
		`getReportQuery()`

Table 2–11 Elements for TopLink Extended API (Cont.)

Element	Default	Method Name

DatabaseQuery		`addArgument(String argumentName)`
		`bindAllParameters()`
		`cacheStatement()`
		`cascadeAllParts()`
		`cascadePrivateParts()`
		`dontBindAllParameters()`
		`dontCacheStatement()`
		`dontCascadeParts()`
		`dontCheckCache()`
		`dontMaintainCache()`
		`dontUseDistinct()`
		`getQueryTimeout()getReferenceClass()`
		`getSelectionCriteria()`
		`refreshIdentityMapResult()`
		`setCall(Call call)`
		`setEJBQLString(String ejbqlString)`
		`setFilter(Expression selectionCriteria)`
		`setQueryTimeout(int queryTimeout)`
		`setSQLString(String sqlString)`
		`setShouldBindAllParameters(boolean shouldBindAllParameters)`
		`setShouldCacheStatement(boolean shouldCacheStatement)`
		`setShouldMaintainCache(boolean shouldMaintainCache)`
		`shouldBindAllParameters()`
		`shouldCacheStatement()`
		`shouldCascadeAllParts()`
		`shouldCascadeParts()`
		`shouldCascadePrivateParts()`
		`shouldMaintainCache()`

JDOTransaction

The `JDOTransaction` class implements the JDO Transaction interface, and defines the basic begin, commit, and rollback APIs, and synchronization callbacks within the UnitOfWork. It supports the optional non-transactional read JDO feature.

Read Modes The read mode of a JDO transaction is set by calling the `setNontransactionalRead()` method.

Note: If the transaction is active when changing read mode, an exception will be thrown.

The read modes are:

Non-Transactional Read Non-transactional reads provide data from the database, but do not attempt to update the database with any changes made to the data when the transaction is committed. This is the default transaction mode from PersistenceManagerFactory. Non-transactional reads support nested Units of Work.

When queries are executed in non-transactional read mode, their results are not subject to the transactional boundary. To update objects from the queries' results, users must modify objects through their transactional instances.

To enable non-transactional read mode, set the non-transactional read flag to true.

Transactional Read Transactional reads provide data from the database and writes any changes to the object back to the database when the transactions commits. When transactional read is used, TopLink uses the same UnitOfWork for all data store interaction (begin, commit, rollback). This can cause the cache to grow very large over time, so this mode should be only used with short-lived PersistenceManager instances to allow the UnitOfWork be garbage collected.

When queries are executed in transactional read mode, their results are transactional instances and they are subject to the transactional boundary. Objects can be updated from the result of a query that is executed in transactional mode.

Because the same UnitOfWork is used in this mode, the transaction is always active and must be released when the read mode is changed from transactional read to non-transactional read.

Caution: It is important to ensure that all changes are committed before calling the TopLink extended API `release()` to release the transaction and its UnitOfWork and setting the non-transactional read mode to true. Failure to do so can result in a loss of the transaction.

To enable transactional read mode, set the non-transactional read flag to false.

Synchronization A Synchronization listener can be registered with the transaction to be notified at transaction completion. The `beforeCompletion` and `afterCompletion` methods are called when the pre-commit and post-commit events of the UnitOfWork are triggered respectively.

Running the TopLink JDO demo

TopLink includes a demo that illustrates some of the JDO functionality. `oracle.toplink.demos.employee.jdo.JDODemo` is based on the project `oracle.toplink.demos.employee.relational.EmployeeProject` and is configured to connect to a Microsoft Access database. The database connection code is in the `applyLogin()` method of the EmployeeProject class. You may have to modify this method if you do not have a Microsoft Access database, or if the connection information for your database is different from what is specified in this code. When the database connection is setup properly, you can start running the JDO demo.

Distributed Cache Synchronization

Within a distributed application environment, the correctness of the data that is available to clients is very important. This issue increases in complexity as the number of servers within an environment increases. To reduce the occurrences of incorrect data ("stale" data) being delivered to clients, TopLink provides a cache synchronization feature. This feature ensures that any client connecting to a cluster of servers is able to retrieve its changes, made through a `UnitOfWork`, from any other server in the cluster (provided that no changes have been made in the interim).

When enabled in a distributed application, changes made in one transaction on a particular node of the application is broadcast to all other nodes within the distributed application. This prevents stale data from spanning transactions and greatly reduces the chance that a transaction will begin with stale data.

Cache Synchronization in no way eliminates the need for an effective locking policy but does reduce the number of Optimistic lock exceptions and can therefore dramatically decrease the amount of work that must be repeated by the application.

Cache synchronization complements the implemented locking policies and can propagate changes synchronously or asynchronously.

Controlling the sessions: the Cache Synchronization Manager

The Cache Synchronization Manager offers several options for controlling the synchronized sessions:

Table 2–12 Properties for CacheSynchronizationManager

Use this code fragment...	To...
`setIsAsynchronous (boolean isAsynchronous)`	Set propagation mode. See "Synchronous versus asynchronous updates" on page 2-59.
`setShouldRemoveConnectionOnError (boolean removeConnection)`	Drop connections in the event of a communication error. See "Error handling" on page 2-59.
`addRemoteConnection (RemoteConnection connection)`	Add new connections to the synchronized cache. See "Advanced options: Managing connections" on page 2-59.
`getRemoteConnections()`	Get remote connections. See "Advanced options: Managing connections" on page 2-59.
`removeAllRemoteConnections()`	'Remove all remote connections from the cache synchronization service. See "Advanced options: Managing connections" on page 2-59.
`removeRemoteConnection (RemoteConnection connection)`	Remove a specific remote connection. See "Advanced options: Managing connections" on page 2-59.
`connectToAllRemoteServers()`	Connect to all servers participating in cache synchronization. See "Deprecated options: Connecting to all remote servers" on page 2-59.

With the Session properties set, the session automatically connects to all other sessions that are on the same network when the session logs in. Any changes made as a result of the session is then broadcast to all other servers on the same network.

Using Cache Synchronization Manager options

The Cache Synchronization Manager enables you to specify two important functions: the update method for the servers in the caching service, and the error handling method used to control communication errors.

Synchronous versus asynchronous updates The CacheSynchronizationManager enables you to specify how other sessions are updated when changes are made on a given node:

- If the changes are sent synchronously, the current transaction does not commit until the changes have been sent.
- If the changes are sent asynchronously, the changes may not have reached all servers before the transaction completes the commit.

Error handling The Cache Synchronization Manager offers very simple error handing: you can set it to drop connections in the event of a communications error.

Advanced options: Managing connections The Cache Synchronization Manager includes several advanced API options for managing connections. These options are listed in Table 2–12, and enable you to get or add connections, as well as remove specific connections or all connections from a cache synchronization service. Note that these options are considered advanced functionality that is not typically required to run a cache synchronization pool.

Deprecated options: Connecting to all remote servers The Cache Synchronization manager continues to support the `connectToAllRemoteServers` functionality. However, this support should be considered only as a service to legacy applications, and not added to new ones.

Using a clustering service

The clustering services for cache synchronization have the following attributes:

Multicast Group IP The IP used by the sessions for multicast communications. All sessions that share the same Multicast group IP and address will send changes to each other

Multicast Port Port used for Multicast communication

Time To Live This is the number of 'hops' that a multicast packet will make on the network before stopping. This has more to do with network configuration than the number of nodes connected

announcementDelay This setting is used by the ClusteringService to determine how long to wait between making the Remote Service available and announcing its existence. This is required in systems where there is propagation delay when binding the services into JNDI.

Implementing a custom Clustering Service You can implement a custom Clustering Service to support cache synchronization. This advanced option must respond and be usable as a Foundation Library supplied Clustering Service. Custom Clustering services, include the all of the components of a regular clustering service (multicast group IP, multicast port, and Time to Live Setting).

Configuring cache synchronization

To configure an TopLink Session to use Cache Synchronization, set the Session to use a Cache Synchronization Manager with a particular Clustering Service in the Session properties. This class controls the interaction with other Sessions including accepting changes and connections from Sessions and sending information to all other Sessions.

The Cache Synchronization Manager requires the URL of the naming service. The Clustering Services are organized by communication framework then by naming service. The implementations shipped with the Foundation Library are as follows:

Table 2–13 Cache Synchronization implementations shipped with the Foundation Library

Name	Naming Service Type	Framework
RMIClusteringService	RMI registry	RMI
RMIJNDIClusteringService	JNDI	RMI
CORBAJNDIClusteringService	JNDI	One for each of Sun, Orbix and Visibroker corba
JMSClusteringService	JNDI	JMS

If you need to implement your own proprietary communications protocol, consult the `RMIRemoteSessionControllerDispatcher` and `RMIRemoteConnection` classes shipped with TopLink.

The Session may also be configured through code.

Example 2–19 Using a simple URL for RMI registry

```
session.setCacheSynchronizationManager(new
oracle.toplink.remote.CacheSynchronizationManager () );
// simple URL used for RMI registry
session.getCacheSynchronizationManager().setLocalHostURL("localhost:1099");
session.getCacheSynchronizationManager().setClusteringServiceClassType(oracle.to
plink.remote.rmi.RMIClusteringService.class);
```

Example 2–20 Setting up clustering with a non-default multicast group

```
session.setCacheSynchronizationManager( new oracle.toplink.remote.
CacheSynchronizationManager());
//simple URL used for RMI registry
session.getCacheSynchronizationManager(). setLocalHostURL("localhost:1099");
// Set up Clustering Service with non default multicast group. Note that the
multicast group must start with 226.x.x.x and can not be 226.0.0.1. The port can
be any value. Set the same multicast IP and port number for all sessions that
you wish to synchronize
RMIClusteringService clusteringService = new RMIClusteringService("226.3.4.5",
3456, session);
session.getCacheSynchronizationManager().setClusteringService(clusteringService)
;
```

Connecting the sessions

To start the framework for synchronizing the sessions:

1. Instantiate a `CacheSynchronizationManager`.
2. Add this manager to the current `ServerSession`.
3. Instantiate a `RemoteSessionDispatcher`, and add this dispatcher to the `CacheSynchronizationManager`.
4. Make the `RemoteSessionDispatcher` available in a global space such as the RMI registry.

   This session is now able to receive synchronization updates and new connections from other servers.

To connect the servers:

1. Create a `RemoteConnection` for the communications framework.
2. Retrieve the `RemoteSessionDispatcher` of a session that is to be synchronized with.
3. Add this `RemoteConnection` to the `CacheSynchronizationManager`.

   The current server connects to the owner session of the dispatcher, and adds that server to the list of servers to synchronize with.

   The distributed session is automatically notified of this session's existence, and adds this session to its list of synchronization participants.

Example 2–21 Adding the RemoteSessionDispatcher to the current Session

```
CacheSynchronizationManager synchManager = new CacheSynchronizationManager();
getSession().setCacheSynchronizationManager(synchManager);
RMIRemoteSessionControllerDispatcher controller = new
oracle.toplink.remote.rmi.RMIRemoteSessionControllerDispatcher(getSession());
synchManager.setSessionRemoteController(controller);
//Lookup and connect to another Session
RemoteConnection connection = new
RMIConnection((RMIRemoteSessionController)registry.lookup("Server2"));
//connect to the distributed session and notify that server of this session's
existence
getSession().getCacheSynchronizationManager().addRemoteConnections(connection);
//Here I am making the current Server available in the Registry your
implementation of distributing the RemoteSessionDispathcer may differ
registry.rebind("Server1", controller);
```

Using Java Messaging Service

Java Messaging Service (JMS) is a a specification that provides developers with a pre-implementation and specification of many common messaging protocols. JMS can also be used to build a more scalable cache synchronization implementation.

TopLink integrates with the JMS publish/subscribe mechanism. For more information on this mechanism, consult the JMS specification available on the Sun web site (http:\www.sun.com).

Preparing to use JMS

A JMS service must be set up outside of TopLink before TopLink can leverage the service. To set the service up the developer must

- Start the JMS service.
- Create a JMS connection factory and note the name. The factory name will be used in TopLink to call the factory.
- Create a JMS topic and note the name. The topic name will be used in TopLink to refer to the topic.

These steps are completed in the software that provides the JMS service. For more information on completing these steps, see the documentation provided with that software.

Setting up JMS in the session configuration file

JMS messaging is typically established in the session configuration file (for example `sessions.xml`) although it can also be set up in code.

The following example illustrates the use of all JMS options in a typical `sessions.xml` file:

```
<cache-synchronization-manager>
    <clustering-service>oracle.toplink.remote.jms
    .JMSClusteringService</clustering-service>
    <should-remove-connection-on-error>false
    </should-remove-connection-on-error>
    <!-- both of the following tags are user specified and must correspond to
    the settings that the user has made, manually, to the JMS Service -->
    <jms-topic-connection-factory-name> TopicConectionFactory
    </jms-topic-connection-factory-name>
    <jms-topic-name> TopLinkCacheSynchTopic
    </jms-topic-name>
    <!-both of the following tags may be required if TopLink is not running in
    the same VM as the JNDI service -->
    <naming-service-url>t3://localhost:7001
    </naming-service-url>
    <naming-service-initial-context-factory>weblogic.jndi.WLInitialContextFactor
    y
    </naming-service-initial-context-factory>
</cache-synchronization-manager>
```

Setting up JMS in Java

JMS support includes new API calls. The following API is required to implement JMS in Java:

```
public void setTopicConnectionFactoryName(String jndiName);
public void setTopicName(String topicName);
```

If the JMS is not running on the same virtual machine as the JNDI service, you may also have to include the following:

```
public void setLocalHostURL(String jndiServiceURL);
public void setInitialContextFactoryName(String initialContextFactoryName);
```

The following example illustrates a typical implementation of JMS:

```
this.session.setCacheSynchronizationManager(new
oracle.toplink.remote.CacheSynchronizationManager());
JMSClusteringService clusteringService = new
oracle.toplink.remote.jms.JMSClusteringService(this.session);
clusteringService.setLocalHostURL("t3://localhost:7001");
```

```
clusteringService.setTopicConnectionFactoryName("TopicConectionFactory");
clusteringService.setTopicName("TopLinkCacheSynchTopic");
this.session.getCacheSynchronizationManager().setClusteringService(clusteringSer
vice);
```

3

Working with Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a component architecture for creating scalable, multi-tier, distributed applications. EJB makes it possible to create dynamically-extensible applications.

This chapter describes TopLink features that provide support for Enterprise JavaBeans. It discusses

- The EJB specification
- Using the session bean model
- Using the entity bean model
- TopLink and container-managed persistent entity beans

In summary, TopLink can be used in conjunction with session beans and bean-managed persistence entity beans designed in accordance with versions 1.0, 1.1, or 2.0 of the Enterprise JavaBean specification.

Container-managed persistence (CMP) for entity beans is provided by specialized products such as TopLink CMP for BEA WebLogic Server and TopLink CMP for IBM WebSphere Server. For information on additional support for other EJB servers, please contact Oracle Support.

The EJB specification

Enterprise JavaBeans (EJBs) represent a standard in enterprise computing developed by Sun Microsystems and its partners. EJB provides a component-based architecture for developing and deploying distributed object-oriented applications in Java. It is important to note the EJB is not itself a product. Enterprise JavaBeans is a specification that describes a framework for developers to use to create distributed business applications and for vendors to use to design application servers. Sun

Microsystems has partnered with industry to specify the EJB framework and many vendors have responded with widespread support for EJB.

Additional information

The EJB home page can be found at www.java.sun.com/products/ejb.

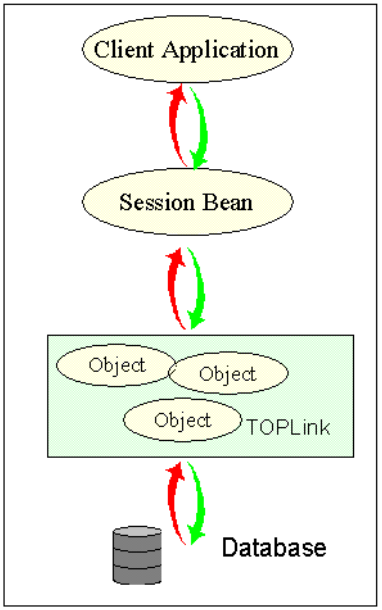
The specification in PDF format is available at www.javasoft.com/products/ejb/docs.html.

Additional information about EJB can also be found at www.javasoft.com/products/ejb/white_paper.html.

Using the session bean model

Session beans are intended to model a process, operation, or service and as such are not themselves persistent. To perform the service they model, session beans can use persistence mechanisms. Under the session bean model, a client application invokes methods on a session bean that in turn performs operations on TopLink-enabled Java objects. All TopLink-related operations are carried out on behalf of the client by the session beans.

Figure 3–1 Basic view of TopLink Session bean architecture



The EJB specification describes session beans as either stateless or stateful.

- Stateful beans maintain a "conversational" state with a client; that is, they retain information between method calls issued by a particular client. Manipulation of persistent objects can occur across multiple method calls. In between those invocations, the EJB Server may decide to passivate (or serialize) the session bean out of the Java VM's memory-space to satisfy some run-time characteristic: performance, memory-footprint, maximum number of beans, and so on. Later, the bean can be activated (un-serialized) and brought back into memory.
- Stateless beans, on the other hand, retain no data between method calls. For stateless session beans, any manipulation of persistent objects must be fully completed within a single method-call.

For more information about Session Beans, activation and passivation, please consult the EJB specification.

Session beans and DatabaseSessions

Both stateful session beans and stateless session beans can be used with a TopLink `DatabaseSession`. However, the way in which TopLink `DatabaseSessions` are handled may be different depending on which type of bean is used.

For stateless beans, no information is stored between method calls from the client, so the bean's connection to the `DatabaseSession` must be re-established for each client method call. Each method call that would require use of TopLink would consist of first obtaining a `DatabaseSession`, making the appropriate calls, and then releasing the reference to the `DatabaseSession`.

For stateful beans, the `DatabaseSession` could be considered a part of the conversational state associated with the bean, and could be retained between calls. However, `DatabaseSession` instances are not fully serializable and will not survive passivation. If the `DatabaseSession` is to be maintained between method calls, it must be released during the passivation process and re-obtained during the activation process.

There are a variety of strategies available for managing session bean access to `DatabaseSessions`. One option available is to make use of the SessionManager class which is described in the following sections.

Interactions with JTS

Many EJB Servers provide a Java Transaction Service (JTS) compliant JDBC driver for use with EJBs. TopLink now has the ability to use such a JTS service. The `DatabaseLogin` must be configured correctly in order to support JTS and session beans.

Please see Chapter 2, "Developing Enterprise Applications" for more information on how to configure TopLink sessions for JTS.

Example 3–1 Configuring DatabaseLogin for interoperability with JTS and EJB

```
DatabaseLogin login = null;
project = null;

//note that useExternalConnectionPooling and useExternalTransactionController
must be set before Session is created
project = new SomeProject();
login = project.getLogin();
login.useExternalConnectionPooling();
login.useExternalTransactionController();
```

```
// usually, other login configuration such as user, password, JDBC URL comes
from the project but these can also be set here
session = new Session(project);

// other session configuration, as necessary: logging, ETC
session.SetExternalTransactionController(new
SomeJTSExternalTransactionController());
session.login();
```

Using session beans with TopLink's three-tier application model

There are several ways to design and implement session beans that make use of TopLink. Here we describe one possible model that uses TopLink's three-tier feature to allow efficient use of database connections. Please refer to Chapter 2, "Developing Enterprise Applications" for a general overview of the TopLink client and server session.

Essentially, the three-tier model can be used to create a `ServerSession` that is to be shared among the various session beans. When the session bean needs to access a TopLink `Session`, the bean obtains a `ClientSession` from the shared `ServerSession`. Using the TopLink three-tier feature provides several benefits:

- *Reduction of overhead*: TopLink project, descriptor, and login information is shared amongst beans
- *Future compatibility with other servers*: all login information is isolated from the beans, which contain no EJB Server-specific information
- *Shared read cache*: efficiency is increased by using a shared cache for reading objects

Using the Session Manager

The TopLink Session Manager is a static utility class that provides developers with any easy way to build a series of sessions that are maintained under a single entity. Use the Session Manager to ensure that a `ServerSession` is always available for any session bean.

The SessionManager class, `oracle.toplink.tools.sessionmanagement.SessionManager`, is contained in the `tl_tools.jar` and can be used to create and manage TopLink sessions defined in an external XML file.

The SessionManager supports the following session types:

- `ServerSession` (for more information, see "Client and server sessions" on page 2-3)
- `DatabaseSession` (for more information, see "Understanding Database sessions" on page 1-2)
- `SessionBroker` (for more information, see "Session broker" on page 2-24

The Session Manager has two main functions: to create instances of these classes, and to ensure that only a single instance of a session with a particular name is returned to the application for any single instance of a SessionManager.

TopLink offers a managed approach for the developer by providing static methods to make a single instance of the SessionManager available globally.

To use this method, instantiate a SessionManager as follows:

```
SessionManager.getManager().
```

Retrieving a session from a SessionManager

The following call is simplest way to retrieve a session from a session manager:

```
getSession(String sessionName)
```

Retrieving a session this way uses all defaults. If no session exists within the current instance of `SessionManager`, the Session Manager attempts to load a session from an XML file called `sessions.xml`. This file, which must be on the root of the class path, contains all of the configuration information for the named session. This allows developers to update the configuration of a session without modifying application code.

> **Note:** While this is the simplest way to retrieve a session, it is not the only way. A session can also be loaded by specifying an XMLLoader to use, the name of the session, the class loader and some flags as defined in the Java Docs. Refer to the Java Docs for specifics on the API that supports this type of custom call.

Using the default configuration file: sessions.xml

By default, SessionManager attempts to create a new session using the specifications contained in a file called `sessions.xml`. This file can define one or more session configurations by name that the SessionManager makes available at runtime. This XML file has a DTD provided that can be found in the installation at `<INSTALL_DIR>\TopLink\core\sessions_4_5.dtd`. The DTD is also documented in Appendix A, "Sessions.xml DTD".

The following is an example `sessions.xml` file. Although it does not show all of the configuration options available it does highlight the most common ones.

```
<?xml version = "1.0" encoding = "US-ASCII"?>
<!DOCTYPE toplink-configuration PUBLIC "-//Oracle Corp.//DTD TopLink for
JAVA//EN" >
<toplink-configuration>
    <session>
        <name>default</name>
        <project-class>MyProject</project-class>
        <session-type><server-session/></session-type>
        <connection-pool>
            <is-read-connection-pool>false</is-read-connection-pool>
            <name>BopPool</name>
            <login>
               <user-name>user</user-name>
               <password>password</password>
            </login>
        </connection-pool>
        <enable-logging>true</enable-logging>
        <logging-options>
            <log-debug>true</log-debug>
            <log-exceptions>true</log-exceptions>
            <log-exception-stacktrace>true</log-exception-stacktrace>
            <print-thread>true</print-thread>
            <print-session>true</print-session>
            <print-connection>true</print-connection>
            <print-date>true</print-date>
        </logging-options>
    </session>
</toplink-configuration>
```

Note that the `DOC_TYPE` indicator is a specific reference to the DTD packaged in the core tl_core.jar. If this entry is not exactly as listed the DTD may not be found.

Using the XMLLoader

The XMLLoader is a mechanism that offers two important features:

- It enables a developer to specify a custom session configuration to be loaded rather than the standard default configuration.
- It enables a developer to control whether or not the session configuration file is re-read when successive sessions are created.

The XMLLoader also enables different sessions, and even different class loaders, to be loaded from different configuration files.

Specifying a custom session configuration file Developers can store customized session configuration information in XML files other than `sessions.xml`. XMLLoader enables the developer to specify this custom configuration file rather than the default `sessions.xml` file. For example:

```
XMLLoader loader = new XMLLoader("MySession.xml");
```

This code creates a new XMLLoader that loads the session configuration from the specified, `MySession.xml`. Note that the file must exist before the `getSession(...)` call is invoked on the SessionManager. By default, the XML file is opened using a ClassLoader's ability to lookup resources as streams. The ClassLoader used for this is the one returned from the default `ConversionManager`.

Reusing the XML file If the XMLLoader instance is maintained, the configuration file is read the first time the `get session ()` call is made but is not reparsed with each subsequent `get session ()` call. If either a different XMLLoader is used to call a session or the API to refresh the configuration file is invoked, the configuration file is be re-parsed, but sessions already in the SessionManager do not change.

Loading Session Broker Using the XMLLoader method to load SessionBroker enables several different sessions from different files to be combined under a single SessionBroker. The developer can make repeated calls to `getSession(XMLLoader, String sessionBrokerName, Class anyObjectClass)`, passing in a different XMLLoader for each file required for the session broker. When all required sessions are loaded in this manner, a completed SessionBroker is returned.

> **Note:** If the developer attempts to load a SessionBroker without all required sessions loaded in to the SessionManager, null is returned.

Example 3–2 Retrieving a named ServerSession

```
SessionManager.getManager().getSession( "employeeSession"); !
```

Example 3–3 A complex call using XMLLoader

```
XMLLoader loader = new XMLLoader("Sessions.xml");
SessionManager.getManager().getSession(loader, "SessionName", SessionBeanClass);
```

Using the entity bean model

Entity beans represent a “business entity.” Entity beans may be shared by many users and are long-lived, able to survive a server failure. Essentially, entity beans *are* persistent data objects – objects with durable state that exist from one moment in time to the next.

TopLink provides a extensive framework for providing developers with Bean Managed Persistence enabled beans. Within the enterprise package TopLink provides a class `oracle.toplink.ejb.bmp.BMPEntityBase`. This class provides developers with a starting point when developing beans. The `BMPEntityBase` class provides implementation for all EJB spec required methods except `ejbPassivate(). ejbPassivate()` is excluded because of special requirements. By sub classing the `BMPEntityBase`, developers have a TopLink enabled entity bean.

To use the `BMPEntityBase`, developers must create the `sessions.xml` file as explained in "Using the Session Manager" on page 3-5. The second requirement is to add a `oracle.toplink.ejb.bmp.BMPWrapperPolicy` to each descriptor that represents an `EntityBean`. This `BMPWrapperPolicy` provides TopLink with the information to create Remote objects for entity beans and to extract the data out of a Remote object. After this is done, the user must create the home and remote interfaces, create deployment descriptors, and deploy the beans.

If a more customized approach is required, TopLink provides a hook into its functionality through the `oracle.toplink.ejb.bmp.BMPDataStore` class. With this class it is possible to easily translate EJB required functionality into simple calls. The `BMPDataStore` provides implementations of load, store, multiple finders and remove functionality. When using the `BMPDataStore`, a `sessions.xml` file is required for use with the SessionManager. A single instance of `BMPDataStore` should exist for each bean type that is deployed within a session. When creating a `BMPDataStore`, pass in the session name of the session that the `BMPDataStore` should use to persist the beans and the class of the Bean type being persisted. Store the `BMPDataStore` in a global location so that each instance of a Bean type uses the correct Store.

If using a customized implementation, the full functionality of the ServerSession and the `UnitOfWork` is available to developers.

TopLink and container-managed persistent entity beans

Because entity beans represent persistent data, bean developers must have some mechanism for making their beans persistent. In most cases this means mapping beans to a relational database. The EJB specification describes a type of entity bean -- container-managed persistent entity bean. For this type of bean, the designer does not have to include calls to any particular persistence mechanism in the bean itself. The EJB Server and its tools use meta-information in the deployment descriptor to describe how the bean is to be persisted to a database. This is commonly referred to as *automatic* persistence.

Also available from Oracle are:

- TopLink CMP for BEA WebLogic Server, a separate product based upon TopLink for Java, that provides container-managed persistent entity beans for the BEA WebLogic Server.
- TopLink CMP for IBM WebSphere Application Server, a separate product based upon TopLink for Java, that provides container-managed persistent entity beans for the IBM WebSphere server.

Please contact Oracle for more information concerning our support of EJB and the use of TopLink for container-managed persistent entity beans.

4

EJBQL Support

Version 2.0 of the EJB specification presents a new query language, called EJBQL. EJBQL is similar to SQL, but differs in that it presents queries from an object model perspective, as opposed to a database perspective.

This chapter discusses the following:

- Why use EJBQL?
- EJBQL structure
- Using EJBQL with TopLink

EJBQL is designed to be compiled to the target language of the persistent data store used by a persistence manager. What differentiates it primarily from SQL is that it includes path expressions that enable navigation over the relationships defined for entity beans and dependent objects. The complete EJB specification, including EJBQL can be found at http://java.sun.com/products/ejb/2.0.html.

Why use EJBQL?

TopLink uses EJBQL to enable users to declare queries using the attributes of each abstract entity bean in the object model. This offers the following advantages:

- There is no need to know the database structure (tables, fields).
- Relationships can be used in a query, via navigation from attribute to attribute.
- Users can construct queries using the attributes of the entity beans instead of using database tables and fields.
- Queries are database-independent so they are portable.
- Users can use “SELECT” to specify the query's reference class (entity bean which you are querying against)

- EJBQL includes support for an attribute level "SELECT" to pick a specific field out of a result set, and answer it.

EJBQL structure

An EJBQL query can contain any of the following components:

FROM clause The `FROM` clause defines the scope of the query. All identification variables used in the rest of the query are defined in this clause. This clause may also contain the key words `IN` and `AS`. Queries must contain a `FROM` clause to be valid.

SELECT clause The `SELECT` defines the return values of the EJBQL query. Return values can be either an attribute, or Entity bean or Java object.

WHERE clause The `WHERE` clause is a conditional expression used to restrict the results of a query. The `WHERE` clause is optional.

A note about notation The examples in this section use a modified Backus-Naur Form (BNF). For more information on BNF, see "About Backus Naur Form" on page D-1.

Basic structure

All EJBQL statements follow the same basic structure:

```
SELECT selectClause FROM fromClause [WHERE whereExpression]
```

The FROM clause

The FROM clause defines the scope of the query by declaring identification variables. The FROM clause designates the domain of the query, which may be constrained by path expressions. This is a mandatory part of the EJBQL statement, and must be in the following syntax:

`FROM` `{identification variabledeclaration}+`
The `{identification variabledeclaration}+` argument resolves to `{AbstractSchemaName entityBeanVariable}`, and may be followed by any number of either of the following:

```
AbstractSchemaName entityBeanVariable, IN(entityBeanVariablePath) [AS]
oneToManyVariable
```

Note: If a FROM clause contains more than one identification variabledeclaration, the expressions must be separated by commas.

This syntax requires entityBeanVariablePath to be specified using the following syntax:

```
entityBeanVariable[.oneToOneRelationshipAttribute]*.oneToManyRelationshipAttribute
```

The FROM clause defined

There are two components to the FROM clause.

FROM {AbstractSchemaName entityBeanVariable} AbstractSchemaName is the name specified as an alias for the entity bean using the tag abstract-schema-name in the ejb-jar.xml file. For example:

```
<abstract-schema-name>EmployeeBean</abstract-schema-name>
```

As indicated, there is always at least one {AbstractSchemaName entityBeanVariable} element, and there may be more. If a FROM clause contains more than one AbstractSchemaName expression, the expressions must be separated by commas

[oneToManyVariable IN entityBeanVariablePath] This sub clause associates the oneToManyVariable with the oneToManyRelationshipAttribute at the end of the entityBeanVariablePath. This element is optional.

Using the FROM clause: a few examples

A simple example The simplest query consists of only select and from clauses:

```
SELECT OBJECT(employee) FROM EmployeeBean employee
```

This query declares employee as a variable representing the EmployeeBean entity bean, and returns all employees in the database.

Using IN to query reference classes The IN keyword designates that the preceding identifier will evaluate to a collection. You can include IN in FROM clause queries to search reference classes:

```
SELECT OBJECT(employee) FROM EmployeeBean employee, IN(employee.phoneNumbers)
phoneNumber
```

In addition to `employee`, this declares `phoneNumber` as a variable representing `PhoneNumber`. This is because `employee.phoneNumbers` is a one-to-many relationship whose reference class is `PhoneNumber`.

```
SELECT OBJECT(employee) FROM EmployeeBean employee,
IN(employee.manager.phoneNumbers) phoneNumber
```

This declares `phoneNumber` as a variable representing `PhoneNumber`. In this case, `manager` is the owner of the one-to-many relationship. This implies that `phoneNumber` will be related to the manager of the employee(s) in the result set, as opposed to the employees themselves.

Using AND The AND operator enables you to combine logical arguments in your query. For example, the following query searches employees with the last name, "Smith", and the phone number area code, "613":

```
SELECT OBJECT(employee) FROM EmployeeBean employee, IN(employee.phoneNumbers)
phoneNumber
WHERE employee.lastName = "Smith" AND
phoneNumber.areaCode = "613"
```

Using AS The `FROM` clause can contain an `AS` used to designate an identifier for the rest of the query. The following two queries are semantically equivalent.

```
SELECT OBJECT(employee) FROM EmployeeBean AS employee WHERE employee.id = 12
SELECT OBJECT(employee) FROM EmployeeBean employee WHERE employee.id = 12
```

The SELECT clause

The SELECT clause defines the types of values to be returned by the query. The return type must be a container-managed relationship (CMR) or a container-managed field (CMF) field for the bean associated with the query.

The SELECT clause defines the types of values to be returned by the query. For TopLink, this defines the reference class and attribute (if specified) returned by the query. It must conform to the following syntax:

```
SELECT OBJECT(<entity bean variable.>)
SELECT entityBeanVariable{.attribute}+
```

Using the SELECT clause: a few examples

A simple example This example returns a collection of `EmployeeBeans`:

```
SELECT OBJECT(employee) from EmployeeBean employee
```

Adding attributes Adding an attribute to the end of the `entityBeanVariable` enables you select only that attribute from the result set. For example, this query returns a collection of the `areaCodes` of the associated `PhoneNumbers`:

```
SELECT phoneNumber.areaCode FROM PhoneNumber phoneNumber
```

Using DISTINCT Adding the `DISTINCT` keyword to a query specifies that the query must eliminate duplicate values from the result set.

```
SELECT DISTINCT OBJECT(employee) FROM EmployeeBean employee
```

The WHERE clause

The WHERE clause is by far the most complex and powerful of the clauses in EJBQL. It is used to define the selection criteria of a query, and consists of a combination of one or more of the following:

- Literals
- Identification variables
- Path expressions
- Input parameters
- Conditional expressions
- Logical
- Boolean and Comparison operators

The Range expressions `IN`, `LIKE` and `BETWEEN` are specified and are modeled on the equivalent SQL behavior. There are also `NULL` tests and several Functional Expressions (`ABS`, `CONCAT`, `LENGTH`, `SQRT`, `SUBSTRING`).

The `WHERE` clause is an optional part of the EJBQL statement, and must be in the following syntax:

```
WHERE conditionalExpression
```

The `WHERE` syntax requires `conditionalExpression` to be defined using the following syntax:

```
conditionalTerm [OR conditionalTerm]*
```

The `conditionalExpression` syntax requires `conditionalTerm` to be defined using the following syntax:

```
conditionalFactor [AND conditionalFactor]*
```

Using constants

The WHERE clause supports the use of String, Integers, Floats, … in defining the selection criteria for the query. Strings are delimited by quotation marks ("<string>").

Comparison Operators

EJBQL supports "=", "<", ">", ">=", "<=", and "<>" comparison operators for arithmetic functions. It also supports "=" and "<>" for non-arithmetic comparisons

= The = operator checks to see if the value or string on the left side of the expression is equal to the value or string on the right. It supports both arithmetic and non-arithmetic comparisons:

Arithmetic example The employee whose id = 25001

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.id = 25001
```

Non arithmetic example Any employee whose first name is Bob

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName = "Bob"
```

< The < operator checks to see if the value on the left side of the expression is less than the value on the right.

Example All employees whose id is less than 25001

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.id < 25001
```

> The > operator checks to see if the value on the left side of the expression is greater than the value on the right.

Example All employees whose id is greater than 25001

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.id > 25001
```

<= The <= operator checks to see if the value on the left side of the expression is less than or equal to the value on the right.

Example All employees whose id is less than or equal to 25001

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.id <= 25001
```

>= The >= operator checks to see if the value on the left side of the expression is greater than or equal to than the value on the right.

Example All employees whose id is greater than or equal to 25001

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.id >= 25001
```

< > The <> operator checks to see if the value on the left side of the expression is not equal to than the value on the right.

Arithmetic example All employees whose id does not equal 25001

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.id <> 25001
```

> **Note:** This can also be represented as:
> `FROM EmployeeBean emp WHERE NOT(emp.id = 25001)`

Non arithmetic example Any employee who does not live in Ottawa

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.address.city <> "Ottawa"
```

Logical operators

EJBQL supports AND, OR, and NOT as logical operators. The precedence order is NOT, AND then OR. This can be modified with brackets which take precedence over all other operators.

AND Use of the AND operator enables you to combine two or more conditions into a single query.

Example This will return all employees with the first name “Sandra” and the last name “Smitty”:

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName = "Sandra" AND
emp.lastName = "Smitty"
```

OR Use of the OR operator enables you to search for records that contain one or more of the specified values or strings. Use of OR does not imply exclusivity; returned records will satisfy at least one of the specified conditions, but may satisfy more.

Example This will return all employees who have an id of 25001 OR 25002

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.id = 25001 OR emp.id = 25002
```

This can be extended to multiple `OR`s

```
FROM EmployeeBean emp WHERE emp.id = 25001 OR emp.id = 25002 OR emp.id = 25003
```

NOT A `NOT` can be added to further modify the query result set by specifying conditions that must not be met by the selected records.

Example The following will return all employees whose first name is not Bob

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE NOT (emp.firstName = "Bob")
```

The query could also have been written as follows:

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName <> "Bob"
```

Figure 4–1 Result Set Using "=" and "NOT" or "< >"

EmployeeDataNotEqual.xls

EMP_ID	F_NAME	L_NAME	START_DATE	B
25002	Sandra	Smitty	1/1/1993	
25003	Sarah	Way	5/1/1995	
25004	John	Smith	1/1/1995	
25005	Mohammad	Sari	1/12/1995	
25006	John	Way	11/11/1991	
25007	Chris	Chan	1/12/1995	
25008	Fred	Jones	1/1/1995	
25009	Nancy	White	1/1/1993	
25010	Jean	Jefferson	1/12/1995	
25011	Betty	Jones	1/1/1995	1

Combining operators Operators can be combined to create more complex queries. For example, the following will return any employees who meet the following criteria:

- They have a first name of John OR
- They have a first name of Bob and a last name of Smith

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName = "John" OR
emp.firstName = "Bob" AND emp.lastName = "Smith"
```

Figure 4–2 Result Set Using "OR"

EMP_ID	F_NAME	L_NAME	START_DATE	E
25001	Bob	Smith	1/1/1993	
25006	John	Way	11/11/1991	
25004	John	Smith	1/1/1995	

This query is slightly different because of the brackets. Only employees who have a last name of Smith with a first name of John or Bob will be returne.d

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE (emp.firstName = "John" OR
emp.firstName = "Bob") AND emp.lastName = "Smith"
```

Figure 4–3 Result Set using "=" and "OR"

EMP_ID	F_NAME	L_NAME	START_DATE
25001	Bob	Smith	1/1/19
25004	John	Smith	1/1/199

Null Comparison Expressions: Null

The null comparison operator enable you to search for records with no content for a specified field.

Example All employees whose first name is not included in the database:

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName IS NULL;
```

Similarly, by adding the `NOT` logical operator, you can search for all employees whose first name appears in the database:

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName IS NOT NULL;
```

Range Expressions

Similar to SQL, EJBQL supports a number of Range expressions. They are `LIKE`, `BETWEEN` and `IN`. You can also modify these expression with `NOT`.

LIKE `LIKE` enables you to use pattern matching to search for records containing a specific patterns. Support for pattern matching for `LIKE` is as follows:

_ signifies that a match must be made for a single character. For example, the expression **12_4** will match **1234** but not **12334**.

% signifies that a match should be made for a range of characters. For example, the expression **12%4** will match **1234**, **1299994** but not **124**.

Examples All employees whose first name starts with "Ji"

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName LIKE "Ji%"
```

Similarly, you can search for All employees whose first name does not start with "Ji"

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName NOT LIKE "Ji%"
```

BETWEEN Lets you choose a contiguous range of numeric values. Always includes the modifier `AND`.

Examples Any employee aged 26 to 36 inclusive

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.age BETWEEN 26 AND 36
```

Similarly, you can search for any employee not aged 55 to 65 inclusive

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.age NOT BETWEEN 55 AND 65
```

IN Lets you specify a group of values used as criteria for the search.

Example You can use `IN` to search for any employee whose salary is a specific amount:

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.salary IN (30000, 40000,
50000)
```

Similarly, you can search for any employee not earning those specific salaries:

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.salary NOT IN (30000, 40000,
50000)
```

Functional Expressions

EJBQL also supports several functions: `CONCAT`; `SUBSTRING`; `LENGTH`; `SQRT`; and `ABS`.

> **Note:** The EJBQL implementation in TopLink does not support the `LOCATE` function because it is not currently supported in the Expression framework.

ABS The ABS operator represents the mathematical absolute value of the selected field.

Example Any employee whose salary's absolute value is 35000

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE ABS(emp.salary) = 35000
```

CONCAT `CONCAT` enables you to combine variables together and search using the result.

Example The full name of any employees whose first name is "John".

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE CONCAT(emp.firstName,
emp.lastName) LIKE "John%"
```

LENGTH `LENGTH` enables you to search for data that is a specific number of characters in length.

Example Any employee whose first name is 5 letters long

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE LENGTH(emp.firstName) = 5
```

SQRT The `SQRT` operator represents the mathematical operatoion, square root. It enables you to search for data the square root of which satisfies some criteria.

Example Any employee whose salary's square root is 200

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE SQRT(emp.salary) = 200
```

SUBSTRING `SUBSTRING` enables you to extract a portion of a given string for use in a `WHERE` clause. `SUBSTRING` includes numeric arguments as follows:

- The first number represents the start position in the string.
- The second number represents the number of characters in the string to be considered.

Example Any employee record for which the first two characters of the `firstName` field are “bo”.

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE SUBSTRING(emp.firstName, 0, 2) =
"Bo"
```

Input Parameters

Input parameters enable you to take advantage of finders written on the home interface of an EJB in which parameters have been specified. Input parameters can be linked to the EJBQL using “`?`” followed by the index (integer) of the required parameter in the finder method.

A simple example A finder with one parameter:

Finder: `employeeHome.findByLastName(lastNameParameter)`

EJBQL: `SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.lastName = ?1`

"`?1`" is replaced at run-time with the `lastNameParameter` passed from the client.

A complex example A finder can contain more than one parameter. For example:

Finder: `employeeHome.findBy(firstName,lastName)`

EJBQL: This finder can accommodate three EJBQL statements:

```
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName =
?1
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.lastName =
?2
SELECT OBJECT(emp) FROM EmployeeBean emp WHERE emp.firstName =
?1 AND emp.lastName = ?2
```

Combining Clauses

You can include all three types of clauses in your queries to make them more effective.

Multiple clauses: a few examples

The following clause returns telephone numbers whose area code are "613":

```
SELECT OBJECT(phone) FROM PhoneNumber phone
    WHERE phone.areaCode = "613"
```

The following return telephone numbers whose area code are "613" and whose employee first name starts with "Bo".

```
SELECT OBJECT(phone) FROM PhoneNumber phone
WHERE phone.areaCode = "613" AND phone.owner.firstName LIKE "Bo%"
```

Using EJBQL with TopLink

EJBQL can be used several different ways in conjunction with TopLink. It may be specified when mapping an object and its attributes to a table via the Mapping workbench. It can also be built and used dynamically at run time via a ReadQuery or the TopLink session.

For information on using EJBQL queries with the TopLink Mapping Workbench, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

ReadAllQuery

The basic API for using a `ReadAll` query with EJBQL is:

```
setEJBQLString("...")
```

A reference class will also be required if no `SELECT` clause is provided. The query can then be executed as any other query would be executed.

Example 1 A simple ReadAllQuery using EJBQL

```
ReadAllQuery theQuery = new ReadAllQuery();
theQuery.setReferenceClass(EmployeeBean.class);
theQuery.setEJBQLString("SELECT OBJECT(emp) FROM EmployeeBean emp");
…
Vector returnedObjects = (Vector)aSession.executeQuery(theQuery);
```

Example 2 A simple ReadAllQuery using EJBQL and passing arguments

The query is defined as in Example 1 but a vector of arguments is created, filled and passed into the executeQuery method

```
// First define the query
ReadAllQuery theQuery = new ReadAllQuery();
theQuery.setReferenceClass(EmployeeBean.class);
theQuery.setEJBQLString("SELECT OBJECT(emp) FROM EmployeeBean emp WHERE
emp.firstName = ?1");
...
// Next define the Arguments
Vector theArguments = new Vector();
theArguments.add("Bob");
...
// Finally execute the query passing in the arguments
Vector returnedObjects = (Vector)aSession.executeQuery(theQuery, theArguments);
```

Session

EJBQL can be executed directly against the session. This will return a Vector of the objects specified by the reference class. The basic API is as follows:

```
aSession.readAllObjects(<ReferenceClass>, <EJBQLCall>)
// <EJBQLCall> is the EJBQL string to be executed and <ReferenceClass> is the
return class type.
```

```
// Call ReadAllObjects on a session.
Vector theObjects = (Vector)aSession.readAllObjects(EmployeeBean.class, new
EJBQLCall( "SELECT OBJECT (emp) from EmployeeBean emp));
```

5

SDK for XML and Non-relational Database Access

A software development kit (SDK) is a programming package that enables a programmer to develop applications for a specific platform. TopLink provides an SDK for non-relational database access and eXtensible Markup Language (XML) support. This chapter discusses the SDK and includes sections on

- Using the TopLink SDK
- Using TopLink XML support

Using the TopLink SDK

The TopLink SDK allows you to extend TopLink to access objects stored on non-relational data stores. To take advantage of the SDK you need to develop a number of classes that can be used by TopLink to access your particular data store. You also need to take advantage of a number of new TopLink Mappings and use many of the customization hooks provided by TopLink that are not used by a typical application that accesses objects stored on a relational database.

There are four major steps to taking advantage of the TopLink SDK:

- Build an Accessor that holds a connection to your non-relational data store.
- Build the TopLink Calls that read/write the appropriate data from/to your non-relational data store. These Calls interact with your data store via the Accessor and convert the data to/from TopLink `DatabaseRows`.
- Build the TopLink Descriptors and Mappings that map your object model to the `DatabaseRows` used by the Calls.
- Connect to the data store and use TopLink to read and write your objects.

Note: You should have a thorough understanding of the base TopLink product and how it interacts with a typical relational database before attempting to customize TopLink to interact with a non-relational data store. The SDK takes advantage of many of the advanced features of TopLink that are not typically used when storing objects in a relational database.

Accessor

If necessary, TopLink uses your implementation of the interface `oracle.toplink.internal.databaseaccess.Accessor` to maintain a "connection" to your non-relational data store. Development of this Accessor is optional – its use is determined by how you decide to gain visibility to a given connection to your data store. For example, instead of using a TopLink Accessor, you could store the connection in a well-known Singleton and have your Calls use that Singleton to gain access to your data store.

If you do not define your own Accessor, the TopLink SDK simply creates an instance of `oracle.toplink.sdk.SDKAccessor` and uses it during execution. The `SDKAccessor` is an implementation of the Accessor interface. It has little or no implementation behind the protocol required by the Accessor interface.

If you do define your own Accessor, you must implement the Accessor interface. You can accomplish this by subclassing `SDKAccessor` and implementing those methods that are supported by your data store, ignoring the others. This is particularly useful if you want TopLink to take advantage of any support for transaction processing offered by your data store.

Data Store Connection

When logging in, a TopLink Session uses your Accessor to establish a connection to your data store by calling the method `connect(DatabaseLogin, Session)`.

The `DatabaseLogin` passed in holds a number of settings, including the user id and password set by your application. See the documentation on `DatabaseLogin` for information on other settings. Other, user-defined, properties can be stored in the `DatabaseLogin` by your application and used by your Accessor to configure its connection.

TopLink occasionally queries the status of your Accessor's connection to your data store by calling the method `isConnected()`. This method returns true if the Accessor still has a connection. It is optional whether the Accessor actually "pings"

your data store to establish the viability of the connection, as this can cause serious performance degradation.

If your Accessor's connection has timed out or been temporarily disconnected, your application can attempt to reconnect by calling the method `reestablishConnection(Session)`. TopLink does not call this method directly – it is called by your application whenever it makes sense for the application to attempt a reconnect.

When logging out, a TopLink Session uses your Accessor to disconnect from your data store by calling the method `disconnect(Session)`.

Call Execution

During execution of your application, the TopLink Session holds on to your Accessor and uses it whenever a Call needs to be executed by calling the method `executeCall(Call, DatabaseRow, Session)`. Typically, the implementation of this `executeCall` method simply logs the activity back to the Session, if necessary, and delegates the actual interaction with the data store to the Call by calling the method `Call.execute(DatabaseRow, Accessor)`, passing itself in as a parameter.

Transaction Processing

If any of your Calls need to be executed together, within the context of a transaction, TopLink indicates to your Accessor that your connection should begin a transaction by calling the method `beginTransaction(Session)`. If any Exceptions occur during the execution of the Calls contained within the transaction, TopLink rolls back the transaction by calling `rollbackTransaction(Session)`. If all the Calls execute successfully, TopLink commits the transaction by calling `commitTransaction(Session)`.

Calls

TopLink Calls are the hooks where TopLink calls out to your code for reading and writing your non-relational data. To write a Call for the TopLink SDK, develop a class that implements the interface `oracle.toplink.sdk.SDKCall` (which extends the interface `oracle.toplink.queryframework.Call`). This requires you to implement a number of methods. Alternatively, you can subclass `oracle.toplink.sdk.AbstractSDKCall` and, at least initially, simply implement a single method, `execute(DatabaseRow, Accessor)`. Most of your development effort will be concentrated on implementing this method.

The outline about assorted Calls is noticeably lacking in sample code, because the code for Calls will be specific to your particular data store.

If you would like to see an example implementation of these Calls, review the code for the XML Calls in the package `oracle.toplink.xml`. These are also discussed in the following section, "Using TopLink XML support" on page 5-28.

At a minimum, you must implement the following calls for every persistent Class that is stored in the non-relational data store:

- Read Object Call
- Read All Call
- Insert Call
- Update Call
- Delete Call
- Does Exist Call

Depending on the capabilities of your data store, you may need to implement any number of the following custom Calls:

- Named Session Call
- Named Descriptor Call

If you want to use TopLink relationship Mappings (for example, `oracle.toplink.tools.workbench.OneToOneMapping` or `oracle.toplink.sdk.SDKObjectCollectionMapping`) you must also implement the appropriate Calls for reading the reference object(s) for each of the Mappings.

If appropriate, any of the calls can be divided into multiple Calls and combined into a single query.

Read Object Call

A Read Object Call reads the data required to build a single object for a specified primary key. The DatabaseRow passed into a Read Object Call is populated with values for the primary key fields for the object to be read from the data store. The Call returns a single DatabaseRow for the object specified by the primary key.

Read All Call

A Read All Call reads the data required to build a collection of *all* the objects (instances) for a particular Class. The DatabaseRow passed into a simple, Class-level Read All Call is empty. The Call returns a collection of all the DatabaseRows for the appropriate Class.

Insert Call

An Insert Call takes a DatabaseRow of the data for a newly created object and inserts it on the appropriate data store. The DatabaseRow passed into an Insert Call contains values for all the mapped fields for the object to be inserted on the data store. The Call returns a count of the number of rows inserted, typically one.

Update Call

An Update Call takes a DatabaseRow of the data for a recently modified object and writes it to the appropriate data store. The DatabaseRow passed into an Update Call is populated with values for the primary key fields for the object to be updated on the data store. The Call's associated ModifyQuery contains another DatabaseRow that contains values for all the mapped fields for the object to be updated on the data store. The Call returns a count of the number of rows updated, typically one.

Delete Call

A Delete Call deletes the data from the appropriate data store for a specified primary key. The DatabaseRow passed into a Delete Call is populated with values for the primary key fields for the object to be deleted from the data store. The Call returns a count of the number of rows deleted, typically one.

Does Exist Call

A Does Exist Call simply checks for the existence of data for a specified primary key. This allows TopLink to determine whether an Insert or Update should be performed for that primary key. The DatabaseRow passed into a Does Exist Call is populated with values for the primary key fields for the object to be inserted or updated on the data store. The Call returns a null if the object does not exist on the data store and a DatabaseRow if the object does exist.

Custom Call

A custom Call can be written for any other capabilities of your non-relational data store. Like a normal TopLink Call, a custom Call can be parameterized. Custom Calls can be stored in named queries in the TopLink DatabaseSession or in any

TopLink Descriptor. The DatabaseRow passed into a Custom Call is populated with values for the parameters defined for the query.

The Call returns whatever is appropriate for the containing query.

Table 5–1 Query types and return values for custom calls

Query	Return value
`DataModifyQuery`	Row count
`DeleteAllQuery`	Row count
`DeleteObjectQuery`	Row count
`InsertObjectQuery`	Row count
`UpdateObjectQuery`	Row count
`DataReadQuery`	Vector of DatabaseRows
`DirectReadQuery`	Vector of DatabaseRows
`ValueReadQuery`	Vector of DatabaseRows
`ReadAllQuery`	Vector of DatabaseRows
`ReadObjectQuery`	DatabaseRow

Database Row

The DatabaseRows that are passed into your Calls and returned by your Calls are like the normal DatabaseRows used by TopLink for relational database activity (these are very similar to hash tables, containing simple key/value pairs), with the additional capability of holding nested DatabaseRows or nested direct values. This allows TopLink to manipulate non-normalized, hierarchical data.

Nested DatabaseRows and direct values are manipulated via a `oracle.toplink.sdk.SDKFieldValue`. Within the TopLink SDK, any field in a DatabaseRow can have a value that is an instance of `SDKFieldValue`. An `SDKFieldValue` can hold one or more nested DatabaseRows or direct values. ("Direct values" are objects that do not have TopLink Descriptors and are typically placed directly into the containing object without any mapping – for example, Strings, Dates, Numbers.) It can also have a data type name indicating the "type" of elements held in the nested collection. Whether this data type name is required is determined by the data store's requirements for nested data elements.

Nested DatabaseRows, themselves, can also contain nested DatabaseRows, and so on. There is no limit to the nesting.

FieldTranslator

There may be times when the names of fields expected by your TopLink Descriptors and DatabaseMappings are different from those generated by your data store, and vice versa. This is particularly true when dealing with aggregate objects. The aggregate Descriptor is defined in terms of a single set of field names. But a number of different AggregateMappings may reference the same aggregate Descriptor, each expecting a different set of field names for the aggregate data. If this is the case, and you are subclassing `oracle.toplink.sdk.AbstractSDKCall`, then you can take advantage of the SDK FieldTranslator to handle this situation. If you are not subclassing AbstractSDKCall, you can still take advantage of the SDK FieldTranslators by building them into your own Calls. Alternatively, you can create your own mechanism for translating field names between TopLink and your data store on a per-Call basis.

The interface `oracle.toplink.sdk.FieldTranslator` defines a simple read and write protocol for translating the field names in a DatabaseRow. The default implementation of this interface, appropriately named `oracle.toplink.sdk.DefaultFieldTranslator`, performs no translations at all.

Another implementation, `oracle.toplink.sdk.SimpleFieldTranslator`, provides a mechanism for translating the field names in a DatabaseRow, either before the row is written to the data store or after the row is read from the data store. SimpleFieldTranslator also allows for wrapping another FieldTranslator and having the read and write translations processed by the wrapped FieldTranslator also.
A SimpleFieldTranslator also translates the field names of any nested DatabaseRows contained in SDKFieldValues.

Building a SimpleFieldTranslator is straightforward.

```
/* Add translations for the first and last name field names. F_NAME on the data
store will be converted to FIRST_NAME for TopLink, and vice versa. Likewise for
L_NAME and LAST_NAME.
*/
AbstractSDKCall call = new EmployeeCall();
SimpleFieldTranslator translator = new SimpleFieldTranslator();
translator.addReadTranslation("F_NAME", "FIRST_NAME");
translator.addReadTranslation("L_NAME", "LAST_NAME");
call.setFieldTranslator(translator);
```

AbstractSDKCall has some convenience methods that allow you to perform the same operation, without building your own translator.

```
AbstractSDKCall call = new EmployeeCall();
    call.addReadTranslation("F_NAME", "FIRST_NAME");
```

```
call.addReadTranslation("L_NAME", "LAST_NAME");
```

If your Calls are all subclasses of AbstractSDKCall, you can take advantage of the convenience methods in SDKDescriptor that sets the same field translations for all the Calls in the DescriptorQueryManager.

```
descriptor.addReadTranslation("F_NAME", "FIRST_NAME");
descriptor.addReadTranslation("L_NAME", "LAST_NAME");
```

SDKDataStoreException

If your Call should encounter a problem while accessing your non-relational data store, it should throw a `oracle.toplink.sdk.SDKDataStoreException` (or a subclass of your own creation). This Exception has state for holding an error code, a Session, an internal Exception, a DatabaseQuery, and an Accessor. An Exception handler can use this state to recover from the thrown Exception or to provide useful information to the user or developer concerning the cause of the Exception.

Descriptors and Mappings

Once you have developed your Calls, you can use them to define the Descriptors and Mappings that TopLink will use to read and write your objects. Instead of using the normal TopLink Descriptors, you will need to use a subclass of Descriptor, `oracle.toplink.sdk.SDKDescriptor`, that provides support for the new mappings supplied by the SDK. Along with the new Mappings that allow non-normalized data to be accessed, most of the typical TopLink Mappings are supported by the SDK.

SDKDescriptor

The TopLink SDK supports most of the properties of the standard Descriptor:

- Basic Properties
- DescriptorQueryManager
- Sequence Numbers
- Inheritance

For more information on other supported and unsupported properties, see "Other supported properties" on page 5-10 and "Unsupported properties" on page 5-10.

Basic Properties The code needed to build a basic `SDKDescriptor` is nearly identical to that used to build a normal Descriptor.

```
SDKDescriptor descriptor = new SDKDescriptor();
descriptor.setJavaClass(Employee.class);
descriptor.setTableName("employee");
descriptor.setPrimaryKeyFieldName("id");
```

The Java class is required. The table name is usually required. Whether you use and/or allow multiple table names will be determined by how the data is stored on your data store and translated by your Calls. It is probably easiest to map the Descriptor to a single table and use your Calls to merge together the data that might be spread across multiple tables into a single table (this would be somewhat analogous to a relational "view"). The primary key field name is also required – it is used by TopLink to maintain object identity.

DescriptorQueryManager The major difference between building an `SDKDescriptor` and building a standard Descriptor is that you need to define *all* the custom Queries for the Descriptor's QueryManager. Typically, to do this, you would build a TopLink DatabaseQuery and put it in the Descriptor's QueryManager.

```
ReadObjectQuery query = new ReadObjectQuery();
    query.setCall(new EmployeeReadCall());
    descriptor.getQueryManager().setReadObjectQuery
    (query);
```

But SDKDescriptor has a number of convenience methods that simplify setting all these Calls.

```
descriptor.setReadObjectCall(new EmployeeReadCall());
descriptor.setReadAllCall(new EmployeeReadAllCall());

descriptor.setInsertCall(new EmployeeInsertCall());
descriptor.setUpdateCall(new EmployeeUpdateCall());
descriptor.setDeleteCall(new EmployeeDeleteCall());

descriptor.setDoesExistCall(new EmployeeDoesExistCall());
```

These Calls are instances of the Calls described in the section "Calls" on page 5-3. In addition to the standard CRUD (Create-Read-Update-Delete) operations represented by these Calls, you can also add custom Calls to an SDKDescriptor that allow your application to query your data store using selection criteria that can be set dynamically.

```
descriptor.addReadAllCall("readByLastName", new EmployeesByLastNameCall(),
"lastName");
descriptor.addReadObjectCall("readByID", new EmployeeByIDCall(), "employeeID");
```

Custom Calls can be invoked by your application at run time with a parameter value that will be passed into the Call via a DatabaseRow. The Call is expected to communicate with your data store and return a DatabaseRow with the appropriate data to build an instance of the appropriate object (in this example, an Employee), as defined by the Mappings in the Descriptor.

Sequence Numbers If your data store provides support for sequence numbers, you can configure your Descriptor to use sequence numbers.

```
descriptor.setSequenceNumberName("employee");
descriptor.setSequenceNumberFieldName("id");
```

To take advantage of sequence numbers you will also need to define a number of custom queries to be used by TopLink for querying and updating the sequence numbers. Custom queries are maintained by the TopLink DatabasePlatform. See the TopLink JavaDocs for more information.

Inheritance The SDKDescriptor supports TopLink inheritance settings.

```
largeProjectDescriptor.setParentClass(Project.class);
```

Whether you configure subclass Descriptors to use tables in addition to the table(s) defined in the superclass Descriptor is determined by how your data store can store the data. Initially, you should try to define a single table in the root class Descriptor and *not* define any additional tables in the subclass Descriptors. Then your Calls can build up DatabaseRows for a single table, simply leaving out the fields that are not required for the particular subclass Descriptor.

Other supported properties The SDKDescriptor supports most other Descriptor properties without any special consideration, namely:

- Interfaces
- Copy Policy
- Instantiation Policy
- Wrapper Policy
- Identity Maps
- Descriptor Events

Unsupported properties There are a few Descriptor properties that are currently unsupported by the TopLink SDK:

- Query Keys
- Optimistic Locking

Standard mappings

The TopLink SDK provides support for many of the DatabaseMappings in the base TopLink class library. In addition to the standard Mappings, the SDK provides four new Mappings that provide support for non-normalized, hierarchical data. For more information, see "SDK Mappings" on page 5-15.

Direct Mappings The TopLink SDK supports all the base TopLink direct mappings:

- "Direct-to-field mappings" on page 7-2
- "Type conversion mappings" on page 7-3
- "Object type mappings" on page 7-4
- "Serialized object mappings" on page 7-6
- "Transformation mappings" on page 7-7

The only Mapping that warrants special consideration is the SerializedObjectMapping. Any Read Calls that support Descriptors that have a SerializedObjectMapping must return the data for the SerializedObjectMapping as either a byte array (`byte[]`) or as a hexadecimal string representation of a byte array. Likewise, TopLink will pass the data for the SerializedObjectMapping to any Write Call as a byte array (`byte[]`).

Relationship Mappings The TopLink SDK provides support for a number of the base TopLink relationship mappings. Any functionality offered by unsupported Mappings can be found in alternative Mappings.

Private relationships The TopLink SDK provides full support for private relationships. Whenever an object is written to the database, its private objects are also written to the database. Likewise, whenever an object is removed from the database, its private objects are also removed.

Your Calls do not need to be aware of private relationships. TopLink will invoke the appropriate Calls to write and delete the private objects when necessary. TopLink determines the appropriate Call for a particular private object by getting it from the object's DescriptorQueryManager.

Indirection The TopLink SDK provides full support for TopLink indirection, in all its various forms (basic, indirect container, and proxy). Indirection can be used to

improve the performance of TopLink relationship mappings by delaying the reading of reference objects until they are actually needed by the original object or any of its client objects.

Your Calls do not need to be aware of indirection. TopLink will invoke the appropriate Call to read in reference objects when they are needed by the application. TopLink determines the appropriate Call for a particular (indirect) relationship by getting the custom selection query from the relationship's Mapping.

Container Policy The TopLink SDK also supports TopLink container policies. A container policy allows you to specify which concrete class TopLink should use for storing query results; whether for a DatabaseQuery or for a CollectionMapping.

Calls do not need to be aware of the container policy. For ease of development, and to support JDK 1.1.x, your Calls simply use a `java.util.Vector` to handle any collection of DatabaseRows. TopLink converts any Vector of DatabaseRows into the appropriate Collection (or Map) of business objects and vice versa. TopLink determines the appropriate concrete container class by getting the container policy from the appropriate DatabaseQuery or DatabaseMapping.

AggregateObjectMapping Due to limitations of the AggregateObjectMapping, the TopLink SDK does not support this Mapping. Nearly equivalent behavior is provided with "SDKAggregateObjectMapping" on page 5-15.

OneToOneMapping The OneToOneMapping is fully supported by the TopLink SDK. You will need to provide the Mapping with a custom selection query.

```
ReadObjectQuery query = new ReadObjectQuery();
query.setCall(new ReadAddressForEmployeeCall());
mapping.setCustomSelectionQuery(query);
```

The Read Call used for the custom selection query will need to be aware of whether the mapping uses a source foreign key or a target foreign key. It will also need to know which field(s) holds the primary and/or foreign key value(s). As a result, it may be useful to construct the Call with the Mapping as parameter (since the Mapping contains this information).

```
query.setCall(new ReadAddressForEmployeeCall(mapping));
```

VariableOneToOneMapping The VariableOneToOneMapping is fully supported by the TopLink SDK. As with the OneToOneMapping, you must provide the Mapping with a custom selection query.

DirectCollectionMapping The DirectCollectionMapping is fully supported by the TopLink SDK. You should use a DirectCollectionMapping if your data store requires you to perform an additional query to fetch the direct values related to a given object.

If the direct values are included, in an hierarchical fashion, within the DatabaseRow for a given object, you should use "SDKDirectCollectionMapping" on page 5-18.

Provide the DirectCollectionMapping with several custom queries. Because the objects contained in a direct collection do not have a Descriptor, you need to provide the mapping with the queries that TopLink uses to insert and delete the reference objects.

The mappings are required in addition to the custom selection query.

```
DirectReadQuery readQuery = new DirectReadQuery();
readQuery.setCall(new ReadResponsibilitiesForEmployeeCall());
mapping.setCustomSelectionQuery(readQuery);

DataModifyQuery insertQuery = new DataModifyQuery();
insertQuery.setCall(new InsertResponsibilityForEmployeeCall());
mapping.setCustomInsertQuery(insertQuery);

DataModifyQuery deleteAllQuery = new DataModifyQuery();
deleteAllQuery.setCall(new DeleteResponsibilitiesForEmployeeCall());
mapping.setCustomDeleteAllQuery(deleteAllQuery);
```

The Mapping does not need a custom update query because, if any of the reference objects change, all of them are simply deleted and re-inserted.

The Read and Delete Calls used for the this Mapping will need to be aware of which field(s) holds the primary key value(s). As a result, it may be useful to construct these Calls with the Mapping as parameter (since the Mapping contains this information).

```
readQuery.setCall(new ReadResponsibilitiesForEmployeeCall(mapping));
deleteAllQuery.setCall(new DeleteResponsibilitiesForEmployeeCall(mapping));
```

OneToManyMapping The OneToManyMapping is fully supported by the TopLink SDK. You should use a OneToManyMapping if, like a typical relational model, the reference objects have foreign keys to the source object (target foreign keys). But if the foreign keys are “forward-pointing” (source foreign keys) and are included, in an hierarchical fashion, within the DatabaseRow for a given object, you should use "SDKAggregateObjectMapping" on page 5-15.

You will need to provide the Mapping with a custom selection query.

```
ReadAllQuery readQuery = new ReadAllQuery();
readQuery.setCall(new ReadManagedEmployeesForEmployeeCall());
mapping.setCustomSelectionQuery(readQuery);
```

Optionally, you can provide the Mapping with a custom delete-all query. If this query is present, TopLink will use it as a performance optimization to delete all the components in the relationship with a single query instead of deleting them one-by-one, when appropriate (for example, when the relationship is private to the containing object).

```
DeleteAllQuery deleteAllQuery = new DeleteAllQuery();
deleteAllQuery.setCall(new DeleteManagedEmployeesForEmployeeCall());
mapping.setCustomDeleteAllQuery(deleteAllQuery);
```

The Read and Delete Calls used for the this Mapping must be aware of which field(s) holds the primary key value(s). As a result, it may be useful to construct these Calls with the Mapping as parameter (since the Mapping contains this information).

```
readQuery.setCall(new ReadManagedEmployeesForEmployeeCall(mapping));
deleteAllQuery.setCall(new DeleteManagedEmployeesForEmployeeCall(mapping));
```

AggregateCollectionMapping The AggregateCollectionMapping is fully supported by the TopLink SDK. The AggregateCollectionMapping is very similar to the OneToManyMapping; but it does not require a “back reference” Mapping from each of the target objects to the source object.

As with the OneToManyMapping, you need to provide the Mapping with a custom selection query and, optionally, a delete-all query.

ManyToManyMapping Because the ManyToManyMapping is very closely tied to the relational implementation of many-to-many relationships, it is not supported by the TopLink SDK. A many-to-many relationship can be mapped with the TopLink SDK by using various combinations of the other collection Mappings (OneToManyMapping, SDKObjectCollectionMapping, etc.).

StructureMapping Because the StructureMapping is tied to the object-relational data model, it is not supported by the TopLink SDK. Nearly identical behavior is provided with "SDKAggregateObjectMapping" on page 5-15.

ReferenceMapping Because the ReferenceMapping is tied to the object-relational data model, it is not supported by the TopLink SDK. Nearly identical behavior can be found in the OneToOneMapping.

ArrayMapping Because the ArrayMapping is tied to the object-relational data model, it is not supported by the TopLink SDK. Nearly identical behavior is provided with "SDKDirectCollectionMapping" on page 5-18.

ObjectArrayMapping Because the ObjectArrayMapping is tied to the object-relational data model, it is not supported by the TopLink SDK. Nearly identical behavior is provided with "SDKDirectCollectionMapping" on page 5-18.

NestedTableMapping Because the NestedTableMapping is tied to the object-relational data model, it is not supported by the TopLink SDK. Nearly identical behavior is provided with "SDKObjectCollectionMapping" on page 5-23.

SDK Mappings

The TopLink SDK provides four new Mappings that provide support for non-normalized, hierarchical data:

- SDKAggregateObjectMapping
- SDKDirectCollectionMapping
- SDKAggregateCollectionMapping
- SDKObjectCollectionMapping

SDKAggregateObjectMapping The SDKAggregateObjectMapping is similar in most ways to the standard AggregateObjectMapping. But there are several differences:

- All the fields used by the reference (aggregate) Descriptor to build the aggregate object are contained in a single, *nested* DatabaseRow, not in the base DatabaseRow. The base DatabaseRow has a single field mapped to the aggregate object attribute that contains an SDKFieldValue. This SDKFieldValue holds the nested DatabaseRow, and this nested DatabaseRow contains all the fields needed by the reference Descriptor to build an instance of the aggregate object.
- There is no need for field name translations. If necessary, the appropriate Call can translate the field names, when it is converting data from the data store's native format to a TopLink DatabaseRow (and vice versa), as described in "FieldTranslator" on page 5-7.

- There is no need for the `isNullAllowed` flag. Since the all the fields used to build the aggregate object are contained in a single field in the base DatabaseRow, there is no need to indicate whether multiple null field values should result in a null object placed in the attribute or a new instance of the aggregate object with all attributes set to null. If the attribute is null, the field value in the base DatabaseRow will be null. If the attribute contains an instance of the aggregate object with all null attributes, the field value in the base DatabaseRow will be an SDKFieldValue with a single, nested DatabaseRow whose field values will all be null.

The code for building an SDKAggregateObjectMapping is similar to that for the AggregateObjectMapping. You need to specify an attribute name, a reference class, and a field name.

```
SDKAggregateObjectMapping mapping = new SDKAggregateObjectMapping();
mapping.setAttributeName("period");
mapping.setReferenceClass(EmploymentPeriod.class);
mapping.setFieldName("period");
descriptor.addMapping(mapping);
```

Because the data used to build the aggregate object is already nested within the base DatabaseRow (in other words, a separate query is not required to fetch the data for the aggregate object), the SDKAggregateObjectMapping does not require any custom queries. But any Read Call that builds the base DatabaseRow to be returned to TopLink must build the DatabaseRow properly. Likewise, any Write Calls must know what to expect in the DatabaseRows passed in by TopLink. Table 5–2 demonstrates an example of the values that would be contained in a typical DatabaseRow with data for an aggregate object.

Table 5–2 Field names and mappings for SDKAggregateObjectMapping

Field Name	Field Value
`employee.id`	1
`employee.firstName`	"Grace"
`employee.lastName`	"Hopper"

Table 5–2 Field names and mappings for SDKAggregateObjectMapping (Cont.)

Field Name	Field Value
employee.period	SDKFieldValue elements=[DatabaseRow(employmentPeriod.startDate="1943-01-01" employmentPeriod.endDate="1992-01-01")] elementDataTypeName="employmentPeriod" isDirectCollection=false

In the example, an SDKAggregateObjectMapping maps the attribute `period` to the field `employee.period` and specifies the reference class as **`EmploymentPeriod`**. The value in the field `employee.period` is an SDKFieldValue with a single, nested DatabaseRow. This nested row will be used by the EmploymentPeriod Descriptor to build the aggregate object. The names of the fields in the nested DatabaseRow must match those expected by the EmploymentPeriod Descriptor.

The code in your Read Calls that builds the DatabaseRow to be returned to TopLink is straightforward.

```
DatabaseRow row = new DatabaseRow();
row.put("employee.id", new Integer(1));
row.put("employee.firstName", "Grace");
row.put("employee.lastName", "Hopper");

DatabaseRow nestedRow = new DatabaseRow();
nestedRow.put("employmentPeriod.startDate", "1943-01-01");
nestedRow.put("employmentPeriod.endDate", "1992-01-01");
Vector elements = new Vector();
elements.addElement(nestedRow);

SDKFieldValue value = SDKFieldValue.forDatabaseRows(elements,
"employmentPeriod");
row.put("employee.period", value);
```

The code in your Write Calls that deconstructs the DatabaseRow generated by TopLink is also straightforward.

```
Integer id = (Integer) row.get("employee.id");
String firstName = (String) row.get("employee.firstName");
String lastName = (String) row.get("employee.lastName");
```

```
SDKFieldValue value = (SDKFieldValue) row.get("employee.period");
DatabaseRow nestedRow = (DatabaseRow) value.getElements().firstElement();
String startDate = (String) nestedRow.get("employmentPeriod.startDate");
String endDate = (String) nestedRow.get("employmentPeriod.endDate");
```

SDKDirectCollectionMapping The SDKDirectCollectionMapping is similar to the standard DirectCollectionMapping in that it represents a collection of objects that are not TopLink-enabled (the objects are not associated with any TopLink Descriptors; for example, Strings). But an SDKDirectCollectionMapping is different from the standard DirectCollectionMapping in that the data representing the collection of objects is nested within the base DatabaseRow – a separate query to the data store is not required to gather up the data, the way it is for a standard DirectCollectionMapping.

The code for building an SDKDirectCollectionMapping is straightforward. You need to specify the attribute and the field names. Optionally, you specify the element data type name. Whether the element audiotape name is required is determined by your data store. If your data store needs something to indicate the "type" of each element in the direct collection, then this setting can be used. Alternatively, this information can be determined by your Call.

```
SDKDirectCollectionMapping mapping = new SDKDirectCollectionMapping();
mapping.setAttributeName("responsibilitiesList");
mapping.setFieldName("responsibilities");
mapping.setElementDataTypeName("responsibility");
//optional
descriptor.addMapping(mapping);
```

The SDKDirectCollectionMapping also has a container policy that allows you to specify the concrete implementation of the Collection interface that holds the direct collection.

```
mapping.useCollectionClass(Stack.class);
```

The SDKDirectCollectionMapping also allows you to specify the Class of objects to be placed in the direct collection or the DatabaseRow. If possible, TopLink will convert the objects contained by the direct collection before setting the attribute in the object or before passing the collection to your Call.

```
// Strings stored on the data store will be converted to Classes and vice versa
mapping.setAttributeElementClass(Class.class);
mapping.setFieldElementClass(String.class);
```

Because the data used to build the direct collection is already nested within the base DatabaseRow (in other words, a separate query is not required to fetch the data for the direct collection), the SDKDirectCollectionMapping does not require any custom queries. But any Read Call that builds the base DatabaseRow to be returned to TopLink must build the DatabaseRow properly. Likewise, any Write Calls must know what to expect in the DatabaseRows passed in by TopLink.

Table 5–3 demonstrates examples of the values that would be contained in a typical DatabaseRow with data for a direct collection.

Table 5–3 Field names and values for SDKAggregateObjectMapping

Field Name	Field Value
`employee.id`	1
`employee.firstName`	"Grace"
`employee.lastName`	"Hopper"

`employee. responsibilities`	`SDKFieldValue`
	`elements=[`
	`"find bugs"`
	`"develop compilers"`
	`]`
	`elementDataTypeName="responsibility"`
	`isDirectCollection=true`

In the example, an SDKDirectCollectionMapping maps the attribute `responsibilitiesList` to the field `employee.responsibilities`. The value in the field `employee.responsibilities` is an SDKFieldValue that contains a collection of Strings that make up the direct collection.

The code in your Read Calls that builds the DatabaseRow to be returned to TopLink is straightforward.

```
DatabaseRow row = new DatabaseRow();
row.put("employee.id", new Integer(1));
row.put("employee.firstName", "Grace");
row.put("employee.lastName", "Hopper");

Vector responsibilities = new Vector();
responsibilities.addElement("find bugs");
responsibilities.addElement("develop compilers");
SDKFieldValue value = SDKFieldValue.forDirectValues(responsibilities,
"responsibility");
```

```
row.put("employee.responsibilities", value);
```

The code in your Write Calls that deconstructs the DatabaseRow generated by TopLink is also straightforward.

```
Integer id = (Integer) row.get("employee.id");
String firstName = (String) row.get("employee.firstName");
String lastName = (String) row.get("employee.lastName");

SDKFieldValue value = (SDKFieldValue) row.get("employee.responsibilities");
Vector responsibilities = value.getElements();
```

SDKAggregateCollectionMapping The SDKAggregateCollectionMapping is more akin to the SDKAggregateObjectMapping than the standard `AggregateCollectionMapping`. The `SDKAggregateCollectionMapping` is used for an attribute that is a collection of aggregate objects that are all constructed from data contained in the base DatabaseRow. (The standard `AggregateCollectionMapping` is more like a OneToManyMapping for a private relationship.)

All the data used by the reference (aggregate) Descriptor to build the aggregate collection is contained in a collection of nested DatabaseRows, not in the base DatabaseRow. The base DatabaseRow has a single field mapped to the aggregate collection attribute that contains an SDKFieldValue. This SDKFieldValue holds the nested DatabaseRows, and these nested DatabaseRows each contain all the fields needed by the reference Descriptor to build a single element in the aggregate collection.

The code for building an SDKAggregateCollectionMapping is similar to that for the SDKAggregateObjectMapping. You need to specify an attribute name, a reference class, and a field name.

```
SDKAggregateCollectionMapping mapping = new SDKAggregateCollectionMapping();
mapping.setAttributeName("phoneNumbers");
mapping.setReferenceClass(PhoneNumber.class);
mapping.setFieldName("phoneNumbers");
descriptor.addMapping(mapping);
```

The SDKAggregateCollectionMapping also has a container policy that allows you to specify the concrete implementation of the Collection interface that holds the direct collection.

```
mapping.useCollectionClass(Stack.class);
```

Because the data used to build the aggregate collection is already nested within the base DatabaseRow (in other words, a separate query is not required to fetch the data for the aggregate collection), the SDKAggregateCollectionMapping does not require any custom queries. But any Read Call that builds the base DatabaseRow to be returned to TopLink must build the DatabaseRow properly. Likewise, any Write Calls must know what to expect in the DatabaseRows passed in by TopLink.

Table 5–4 demonstrates examples of the values that would be contained in a typical DatabaseRow with data for an aggregate collection.

Table 5–4 Field names and values for SDKAggregateCollectionMapping

Field Name	Field Value
`employee.id`	1
`employee.firstName`	"Grace"
`employee.lastName`	"Hopper"

`employee. phoneNumbers`	`SDKFieldValue`
	`elements=[`
	`DatabaseRow(`
	`phone.areaCode="888"`
	`phone.number="555-1212"`
	`phone.type="work"`
	`)`
	`DatabaseRow(`
	`phone.areaCode="800"`
	`phone.number="555-1212"`
	`phone.type="home"`
	`)`
	`]`
	`elementDataTypeName="phone"`
	`isDirectCollection=false`

In the example, an SDKAggregateCollectionMapping maps the attribute `phoneNumbers` to the field `employee.phoneNumbers` and specifies the reference class as `PhoneNumber`. The value in the field `employee.phoneNumbers` is an SDKFieldValue with a collection of nested DatabaseRows. These nested rows are used by the PhoneNumber Descriptor to build the elements of the aggregate collection. The names of the fields in the nested DatabaseRows must match those expected by the PhoneNumber Descriptor.

The code in your Read Calls that builds the DatabaseRow to be returned to TopLink is straightforward.

```
DatabaseRow row = new DatabaseRow();
row.put("employee.id", new Integer(1));

row.put("employee.firstName", "Grace");
row.put("employee.lastName", "Hopper");

Vector elements = new Vector();

DatabaseRow nestedRow = new DatabaseRow();
nestedRow.put("phone.areaCode", "888");
nestedRow.put("phone.number", "555-1212");
nestedRow.put("phone.type", "work");
elements.addElement(nestedRow);

nestedRow = new DatabaseRow();
nestedRow.put("phone.areaCode", "800");
nestedRow.put("phone.number", "555-1212");
nestedRow.put("phone.type", "work");
elements.addElement(nestedRow);

SDKFieldValue value = SDKFieldValue.forDatabaseRows(elements, "phone");
row.put("employee.phoneNumbers", value);
```

The code in your Write Calls that deconstructs the DatabaseRow generated by TopLink is also straightforward.

```
Integer id = (Integer) row.get("employee.id");
String firstName = (String) row.get("employee.firstName");
String lastName = (String) row.get("employee.lastName");

SDKFieldValue value = (SDKFieldValue) row.get("employee.phoneNumbers");
Enumeration enum = value.getElements().elements();
while (enum.hasMoreElements()) {DatabaseRow nestedRow = (DatabaseRow)
enum.nextElement();
String areaCode = (String) nestedRow.get("phone.areaCode");
String number = (String) nestedRow.get("phone.number");
String type = (String) nestedRow.get("phone.type");
// do stuff with the values
}
```

SDKObjectCollectionMapping The SDKObjectCollectionMapping is similar to the standard OneToManyMapping, with one important difference. While the standard OneToManyMapping is used to map a collection of target objects that are stored on the database with foreign keys pointing back to the source object's primary key, the SDKObjectCollectionMapping is used to map a collection of target objects that are constructed from a collection of foreign keys contained in the base DatabaseRow that reference the target objects' primary keys. In other words, the foreign keys in a OneToManyMapping are “back-pointing”; the foreign keys in an SDKObjectCollectionMapping are “forward-pointing”.

All the foreign keys used by the mapping to reference the target objects are contained in a collection of *nested* DatabaseRows, not in the base DatabaseRow. The base DatabaseRow has a single field mapped to the object collection attribute that contains an SDKFieldValue. This SDKFieldValue holds the nested DatabaseRows, and these nested DatabaseRows each contain all the fields needed to build a foreign key to an element object's primary key.

The code for building an SDKObjectCollectionMapping is similar to that for the OnetoManyMapping. You need to specify an attribute name, a reference class, a field name, and the source foreign key/target key relationships. Optionally, you specify the reference data type name. Whether the reference data type name is required is determined by your data store. If your data store needs something to indicate the “type” of each reference in the collection of foreign keys, then this setting can be used. Alternatively, this information can be determined by your Call. Because a separate query is required to read in the reference objects contained in the collection, you must build a custom selection query.

```
SDKObjectCollectionMapping mapping = new SDKObjectCollectionMapping();
mapping.setAttributeName("projects");
mapping.setReferenceClass(Project.class);
mapping.setFieldName("projects");
mapping.setSourceForeignKeyFieldName("projectId");
mapping.setReferenceDataTypeName("project"); // optional
mapping.setSelectionCall(new ReadProjectsForEmployeeCall());
descriptor.addMapping(mapping);
```

The SDKObjectCollectionMapping also has a container policy that allows you to specify the concrete implementation of the Collection interface that holds the collection of objects.

```
mapping.useCollectionClass(Stack.class);
```

Any Read Call that builds the base DatabaseRow to be returned to TopLink must build the DatabaseRow properly. Likewise, any Write Calls must know what to expect in the DatabaseRows passed in by TopLink. Table 5–5 demonstrates an example of the values that would be contained in a typical DatabaseRow with data for a collection of foreign keys.

Table 5–5 Field names and values for SDKObjectCollectionMapping

Field Name	Field Value
`employee.id`	1
`employee.firstName`	"Grace"
`employee.lastName`	"Hopper"

`employee.projects`	`SDKFieldValue`
	`elements=[`
	`DatabaseRow(`
	`project.projectId=42`
	`)`
	`DatabaseRow(`
	`project.projectId=17`
	`)`
	`]`
	`elementDataTypeName="project"`
	`isDirectCollection=false`

In the example, an SDKObjectCollectionMapping maps the attribute `projects` to the field `employee.projects` and specifies the reference class as `Project`. The value in the field `employee.projects` is an SDKFieldValue with a collection of nested DatabaseRows.

Nested rows contain foreign keys that will be used by the Mapping's custom selection query to read in the elements of the object collection. The names of the fields in the nested DatabaseRows must match those expected by the custom selection query's Call.

The code in your Read Calls that builds the DatabaseRow to be returned to TopLink is straightforward.

```
DatabaseRow row = new DatabaseRow();
ow.put("employee.id", new Integer(1));
row.put("employee.firstName", "Grace");
row.put("employee.lastName", "Hopper");

Vector elements = new Vector();
```

```
DatabaseRow nestedRow = new DatabaseRow();
nestedRow.put("project.projectId", new Integer(42));
elements.addElement(nestedRow);

nestedRow = new DatabaseRow();
nestedRow.put("project.projectId", new Integer(17));
elements.addElement(nestedRow);

SDKFieldValue value = SDKFieldValue.forDatabaseRows(elements, "project");
row.put("employee.projects", value);
The code in your Write Calls that deconstructs the DatabaseRow generated by
TopLink is also straightforward.
Integer id = (Integer) row.get("employee.id");
String firstName = (String) row.get("employee.firstName");
String lastName = (String) row.get("employee.lastName");

SDKFieldValue value = (SDKFieldValue row.get("employee.projects");
Enumeration enum = value.getElements().elements();
while (enum.hasMoreElements(DatabaseRow nestedRow = (DatabaseRow)
enum.nextElement();
Object projectId = nestedRow.get("project.projectId");
// do stuff with the foreign key
}
```

Sessions

After you have developed your Accessor and your Calls and have mapped your object model to your data store, you can configure and log in to a DatabaseSession and read and write your objects. There are several steps to configuring and logging in to a DatabaseSession for the TopLink SDK:

- If necessary, build an instance of your custom Platform.
- Build an instance of SDKLogin, with this Platform if necessary.
- Build up a TopLink Project with this Login, populating it with your Descriptors.
- Acquire a Session from this TopLink Project and log in.

SDKPlatform

If you are using sequence numbers and you would like TopLink to manage them for you, you may need to create your own subclass of `oracle.toplink.sdk.SDKPlatform`. If you are not using sequences numbers, you can simply use the default behavior in SDKPlatform and ignore this section.

TopLink uses the Platform classes to isolate the database platform-specific implementations of two major activities:

- SQL generation
- Sequence number generation

Since the TopLink SDK is generally unconcerned with SQL generation, probably the only reason you might want to develop your own Platform is if your data store provided a mechanism for generating sequence numbers. If this is the case, you will need to create your subclass and override the appropriate methods for building the Calls that will read and update sequence numbers.

The sequence number Read Call should be built and returned by the method `buildSelectSequenceCall()`. This Call will be invoked by TopLink when TopLink needs to read the value of a specific sequence number. The DatabaseRow passed into the Call will contain one field: the field name will be the `sequenceNameFieldName` (as set in the SDKLogin); the field value will be the name of the sequence number whose current value should be returned by the Call.

The sequence number Update Call should be built and returned by the method `buildUpdateSequenceCall()`. This Call will be invoked by TopLink when TopLink needs to update the value of a specific sequence number. The DatabaseRow passed into the Call will contain two fields:

- The first field name will be the `sequenceNameFieldName` (as set in the SDKLogin); the field value will be the name of the sequence number whose value should be updated by the Call.
- The second field name will be the `sequenceCounterFieldName` (again, as set in the SDKLogin); the field value will be the new value of the sequence number identified by the first field.

SDKLogin

Once you have established whether you need a custom Platform, you can construct and configure your SDKLogin with it.

```
SDKLogin login = new SDKLogin(new EmployeePlatform());
```

If you do not need a custom Platform, you can simply use the default constructor for SDKLogin.

```
SDKLogin login = new SDKLogin();
```

If you are using a custom Accessor to maintain a connection to your data store, you will need to configure the Login to use it. This will allow TopLink to construct a

new instance of your Accessor whenever a connection to the data store is required. If you are not using a custom Accessor, you do not need to set this property, and the Login will be configured to use the SDKAccessor class by default.

```
login.setAccessorClass(EmployeeAccessor.class);
```

After these settings are configured, you can configure the values of the more standard Login properties.

```
login.setUserName("user");
login.setPassword("password");

login.setSequenceTableName("sequence");
login.setSequenceNameFieldName("name");
login.setSequenceCounterFieldName("count");
```

You can also store other, non-TopLink-related properties, in the Login. These properties can be used by your custom Accessor when it connects to the data store.

```
login.setProperty("foo", aFoo);
Foo anotherFoo = (Foo) login.getProperty("foo");
```

TopLink Project

After you have configured your Login, you can build your TopLink Project. You create an instance of `oracle.toplink.sessions.Project`, passing it your Login. Then you add your Descriptors to the Project.

```
Project project = new Project(login);
project.addDescriptor(buildEmployeeDescriptor());
project.addDescriptor(buildAddressDescriptor());
project.addDescriptor(buildProjectDescriptor());
// etc.
```

Session

Finally, after you have your TopLink Project built, you can obtain a DatabaseSession (or ServerSession) and log in.

```
DatabaseSession session = project.createDatabaseSession();
session.login();
```

Now you can use the Session to query for objects, acquire a UnitOfWork, modify objects, and so on.

```
Vector employees = session.readAllObjects(Employee.class);
Employee employee = (Employee) employees.firstElement();
UnitOfWork uow = session.acquireUnitOfWork();
```

```
Employee employeeClone = uow.registerObject(employee);
employeeClone.setSalary(employeeClone.getSalary() + 50);
uow.commit();
```

When you are finished with the Session, you can log out.

```
session.logout();
```

Unsupported features

Currently, there are three major Session features that are unsupported by the TopLink SDK:

- Expressions. Although TopLink Expressions are not supported, there is no limit to the flexibility of your custom Read Calls. The selection criteria for these Calls are only limited by the capabilities of your data store. The selection criteria can be parameterized by adding arguments to the Read Queries.
- Pessimistic Locking
- Cursored Streams and Scrollable Cursors

Using TopLink XML support

TopLink enables you to read and write objects from and to XML files. In fact, TopLink itself reflectively uses this capability to store the Descriptors, Mappings, and other objects that make up a TopLink Project. This capability to perform Object-XML (O-X) Mapping allows your applications to deal exclusively with objects instead of having to deal with the intricacies of XML parsing and deconstruction. This can be particularly helpful for applications that deal with exchanging data with other applications (for example, legacy applications or business partner applications).

Getting Started

There are not many differences between configuring your application to use standard TopLink and configuring it to use the XML extension in its default configuration.

Configure your Login using an XMLFileLogin.

```
XMLFileLogin login = new XMLFileLogin();
login.setBaseDirectoryName("C:\Employee Database");

// set up the sequences
```

```
login.setSequenceRootElementName("sequence");
login.setSequenceNameElementName("name");
login.setSequenceCounterElementName("count");

// create the directories if they don't already exist
login.createDirectoriesAsNeeded();
```

Build your Project.

```
Project project = new Project(login);
project.addDescriptor(buildEmployeeDescriptor());
project.addDescriptor(buildAddressDescriptor());
project.addDescriptor(buildProjectDescriptor());
// etc.
```

Build your Descriptors using XMLDescriptors.

```
XMLDescriptor descriptor = new XMLDescriptor();
escriptor.setJavaClass(Employee.class);
descriptor.setRootElementName("employee");
descriptor.setPrimaryKeyElementName("id");
descriptor.setSequenceNumberName("employee");
descriptor.setSequenceNumberElementName("id");
// etc.
```

Build your Mappings

Limit yourself, at least initially, to using the standard Direct Mappings and the following relationship Mappings:

- OneToOneMapping
- VariableOneToOneMapping
- SDKAggregateObjectMapping
- SDKDirectCollectionMapping
- SDKAggregateCollectionMapping
- SDKObjectCollectionMapping

For the XML extension, OneToOneMappings and SDKObjectCollectionMappings require custom selection queries:

```
// 1:1 mapping
OneToOneMapping addressMapping = new OneToOneMapping();
addressMapping.setAttributeName("address");
```

```
addressMapping.setReferenceClass(Address.class);
addressMapping.privateOwnedRelationship();
addressMapping.setForeignKeyFieldName("addressId");
// build the custom selection query
ReadObjectQuery addressQuery = new ReadObjectQuery();
addressQuery.setCall(new XMLReadCall(addressMapping));
addressMapping.setCustomSelectionQuery(addressQuery);
descriptor.addMapping(addressMapping);
// 1:n mapping

SDKObjectCollectionMapping projectsMapping = new SDKObjectCollectionMapping();
projectsMapping.setAttributeName("projects");
projectsMapping.setReferenceClass(Project.class);
projectsMapping.setFieldName("projects");
projectsMapping.setSourceForeignKeyFieldName("projectId");
projectsMapping.setReferenceDataTypeName("project");
// use convenience method to build the custom selection query
projectsMapping.setSelectionCall(new XMLReadAllCall(projectsMapping));
descriptor.addMapping(projectsMapping);
```

Build your DatabaseSession and log in.

```
DatabaseSession session = project.createDatabaseSession();session.login();
```

Build your sequences, if necessary.

```
(new XMLSchemaManager(session)).createSequences();
```

Use the Session

You can now use the session to query for objects, acquire a UnitOfWork, modify objects, and so on.

```
Vector employees = session.readAllObjects(Employee.class);
Employee employee = (Employee) employees.firstElement();
UnitOfWork uow = session.acquireUnitOfWork();
Employee employeeClone = uow.registerObject(employee);
employeeClone.setSalary(employeeClone.getSalary() + 50);
uow.commit();
```

When you are finished with the Session, you can log out.

```
session.logout();
```

Customizations

There are two main areas of the XML extension that can be customized in a straightforward fashion:

- If you want to change where and/or how the XML documents are stored, you will need to develop your own implementation of the XMLAccessor interface.
- If you want to change how XML documents are translated into DatabaseRows, and vice versa, you will need to develop you own implementation of the XMLTranslator interface.

Implementation details

The classes that implement the support for O-X mapping are in the package `oracle.toplink.xml`. These classes actually make up a simple example of how to use the TopLink SDK as described in the previous section. In addition to implementing the various SDK interfaces, the XML package defines its own set of interfaces that you can implement to slightly alter how your objects are mapped to XML documents without re-implementing the entire SDK suite of interfaces and subclasses.

The XML extension has its own set of implementations of the various SDK interfaces and subclasses:

- XMLFileAccessor
- XMLCall
- XMLDescriptor
- XMLPlatform
- XMLFileLogin
- XMLSchemaManager

The XML extension also defines it own set of interfaces that allow you to plug in your own implementation classes to easily alter the way your objects are mapped to XML documents:

- XMLAccessor
- XMLTranslator
- XML Zip File Extension

XMLFileAccessor

The XMLFileAccessor is a subclass of the SDKAccessor that defines how XML documents are stored in a native file system. As a subclass of SDKAccessor, the XMLFileAccessor is not required to implement any of the Accessor protocol; and, in fact, it only implements the method connect(DatabaseLogin, Session). The XMLFileAccessor uses the standard SDK method of Call execution, and does not support transaction processing, which is a limitation typical of native file systems.

XMLAccessor implementation

In addition to the Accessor interface, the XMLFileAccessor implements the XMLAccessor interface. The XMLAccessor interface defines the protocol necessary for fetching Streams of data for reading and writing XML documents. The XMLFileAccessor implements this protocol by wrapping Files in Streams that can be used by the XMLCalls to read or write XML documents.

The XMLAccessor methods defined for fetching a Stream (either a `java.io.Reader` or `java.ioWriter`) typically requires three parameters:

- A root element name
- A DatabaseRow
- A Vector of DatabaseFields (the ordered primary key element names)

The XMLFileAccessor takes the values of these three parameters and resolves them to a File that will be wrapped with a Stream (either a `java.io.FileReader` or a `java.io.FileWriter`) to be returned to the XMLCall for processing. The File name is calculated in the following fashion:

- The base directory is determined by the configuration of the XMLFileLogin. The base directory, as handled by TopLink, is analogous to a relational database that contains a collection of related tables. If the base directory name is not specified, the current working directory is used (for example, `C:\EmployeeDB`).
- The subdirectory will have the same name as the XML root element name. The root element name, as handled by TopLink, is analogous to the table name in the relational model. In other words, all the XML documents in the same directory will have the same root element name (for example, `C:\EmployeeDB\employee`).
- The file name root is determined by the Vector of DatabaseFields and the DatabaseRow. The filename, as handled by TopLink, is analogous to a row within a table in the relational model. The Vector indicates which fields in the DatabaseRow make up the primary key. The values in these fields in the

DatabaseRow (which should all be Strings at this point) are concatenated together in the same order as they are listed in the Vector. This composite String forms the root of the file name (for example, `C:\EmployeeDB\employee\1234`).

- The file name extension is determined by the configuration of the XMLFileLogin. This extension is optional and arbitrary - in some cases it simply allows the native file system to associate other applications with the files if necessary. If the file name extension is not specified, the default is ".xml" (for example, `C:\EmployeeDB\employee\1234.xml`).

Directory creation

The XMLFileAccessor has one other setting that is configured via the XMLFileLogin: `createsDirectoriesAsNeeded`. If this property is set to true, the Accessor will lazily create directories as they are required, including the base directory. If this property is set to false, which is the default, the Accessor throw an XMLDataStoreException if it encounters a request for an XML document that resolves to a file in a directory that does not exist.

XMLCall

XMLCall and its subclasses are the layer between the Call interface used by TopLink DatabaseQueries and the XML document accessing protocol provided by an XMLAccessor. The XMLFileAccessor implements the XMLAccessor protocol and is used by the XMLCalls; while the XMLCalls implement the Call interface and are used by the standard TopLink DatabaseQueries. The DatabaseQueries are used by your client application and your Descriptors to read and write objects.

All the XMLCalls have two properties in common:

- XMLStreamPolicy
- XMLTranslator

XMLStreamPolicy

The XMLStreamPolicy is yet another interface that defines a protocol for fetching Streams of data for reading and writing XML documents. The default implementation used by the XMLCalls is XMLAccessorStreamPolicy. This implementation simply delegates every request for a Stream to the XMLAccessor. This policy allows the default behavior to be overridden on a per-Call basis. For example, in certain situations, you might want to specify a specific File that holds an XML document instead of relying on the XMLFileAccessor to resolve which File

to use. (In fact, this behavior is already provided by XMLFileStreamPolicy and supported by the methods `XMLCall.setFile(File)` and `XMLCall.setFileName(String)`.)

XMLTranslator

The XMLTranslator is the object used by the XMLCalls to translate an XML document into a TopLink DatabaseRow and vice versa. This is another pluggable interface that allows you to modify the behavior of the XMLCalls. The XMLCalls' default implementation of XMLTranslator is DefaultXMLTranslator.

XMLTranslator implementations

There are a number of subclasses of XMLCall that provide concrete implementations of Call (and SDKCall). The main difference among these classes is their respective implementations of the method `Call.execute(DatabaseRow, Accessor)`.

Six of these subclasses are used for manipulating objects:

- XMLReadCall
- XMLReadAllCall
- XMLInsertCall
- XMLUpdateCall
- XMLDeleteCall
- XMLDoesExistCall

Four subclasses are used to manipulate un-mapped DatabaseRows (for example, raw data):

- XMLDataReadCall
- XMLDataInsertCall
- XMLDataUpdateCall
- XMLDataDeleteCall

Object-Level Calls With a few exceptions, the following object-level Calls all require an association with a DatabaseQuery to operate successfully. This happens automatically when you build a DatabaseQuery and configure it to use a custom call, which is required when using the TopLink SDK and/or the TopLink XML

extension. The Read Calls that are associated with a relationship Mapping do not require an associated DatabaseQuery.

XMLReadCall Given an object-level DatabaseQuery or a OneToOneMapping, an XMLReadCall gets the appropriate XML document and converts it to a DatabaseRow to be mapped to the appropriate object. If the XMLReadCall has a reference to a OneToOneMapping, it will extract the foreign key for the Mapping's relationship from the DatabaseRow passed in to the method `execute(DatabaseRow, Accessor)`. If no Mapping is present, the XMLReadCall extracts the primary key for the Query's associated Descriptor from the DatabaseRow. This key is then used to find the appropriate XML document.

XMLReadAllCall Given an object-level DatabaseQuery or an SDKObjectCollectionMapping, an XMLReadAllCall gets the appropriate XML documents and converts them to a Vector of DatabaseRows to be mapped to the appropriate objects. If the XMLReadAllCall has a reference to an SDKObjectCollectionMapping, it extracts the foreign keys for the Mapping's relationship from the DatabaseRow passed in to the method `execute(DatabaseRow, Accessor)`. The foreign keys are then used to find the appropriate XML documents. If no Mapping is present, the XMLReadAllCall determines the root element name for the Query's associated Descriptor and returns all the DatasebaseRows for that root element name (a true read all).

XMLInsertCall An XMLInsertCall takes the DatabaseRow passed in to the method `execute(DatabaseRow, Accessor)` and uses the primary key in it to find the appropriate XML document Stream. It then takes the “modify row” from the associated ModifyQuery, converts it to an XML document, and writes it out.

If the XML document already exists, an XMLDataStoreException is thrown.

XMLUpdateCall Like an XMLInsertCall, an XMLUpdateCall takes the DatabaseRow passed in to the method `execute(DatabaseRow, Accessor)` and uses the primary key in it to find the appropriate XML document Stream. It then takes the “modify row” from the associated ModifyQuery, converts it to an XML document, and writes it out.

If the XML document does not already exist, an XMLDataStoreException is thrown.

XMLDeleteCall An XMLDeleteCall takes the DatabaseRow passed in to the method `execute (DatabaseRow, Accessor)` and uses the primary key in it to find the appropriate XML document Stream. It then deletes this Stream.

If the XML document already existed, the Call returns a row count of one; if not, the Call returns a row count of zero.

XMLDoesExistCall An XMLDoesExistCall takes the DatabaseRow passed in to the method `execute(DatabaseRow, Accessor)` and uses the primary key in it to find the appropriate XML document Stream. If the document exists, it is converted to a DatabaseRow that can be used to verify the object's existence; otherwise a null is returned.

Data Calls Because XMLDataCalls are not associated with a DatabaseQuery, like the "Object-Level Calls" on page 5-34, a bit more up-front configuration is required. Every XMLDataCall requires a root element name and a set of ordered primary key element names. At run time, these settings are passed to the XMLStreamPolicy (and, usually, on to the XMLFileAccessor), along with the appropriate DatabaseRow, to determine the appropriate XML document Stream.

```
XMLDataReadCall call = new XMLDataReadCall();
call.setRootElementName("employee");
call.setPrimaryKeyElementName("id");
```

XMLDataReadCall An XMLDataReadCall takes the DatabaseRow passed in to the method `execute (DatabaseRow, Accessor)` and uses the primary key in it to find the appropriate XML document Stream and convert it to a DatabaseRow. To provide a consistent result object, this single DatabaseRow is returned inside a Vector.

If the XMLDataReadCall does not have any primary key element names set, it performs a simple read-all for all the XML documents with the specified root element name. These are converted and returned as a Vector of DatabaseRows.

XMLDataReadCalls can be further configured to specify which fields in the resulting DatabaseRow(s) should be returned and what their types should be.

```
XMLDataReadCall call = new XMLDataReadCall();
call.setRootElementName("employee");
call.setPrimaryKeyElementName("id");
call.setResultElementName("salary");
call.setResultElementType(java.math.BigDecimal.class);
```

XMLDataInsertCall An XMLDataInsertCall takes the DatabaseRow passed in to the method `execute(DatabaseRow, Accessor)` and uses the primary key in it to find the appropriate XML document Stream. It then takes that same row, converts it to an XML document, and writes it out.

If the XML document already exists, an XMLDataStoreException is thrown.

XMLDataUpdateCall An XMLDataUpdateCall takes the DatabaseRow passed in to the method `execute(DatabaseRow, Accessor)` and uses the primary key in it to find the appropriate XML document Stream. It then takes that same row, converts it to an XML document, and writes it out.

If the XML document does not already exist, an XMLDataStoreException is thrown.

XMLDataDeleteCall An XMLDataDeleteCall takes the DatabaseRow passed in to the method `execute(DatabaseRow, Accessor)` and uses the primary key in it to find the appropriate XML document Stream. It then deletes this Stream.

If the XML document already existed, the Call returns a row count of one; if not, the Call returns a row count of zero.

XMLDescriptor

XMLDescriptor is a subclass of SDKDescriptor that adds two bits of helpful behavior:

- It automatically initializes its QueryManager with a set of default DatabaseQueries that are already configured to use the appropriate XMLCalls. If you are using TopLink's default support for XML documents, no further modification of these Calls is required.
- It adds methods that are named more in accordance with XML concepts than relational ones. The method `setRootElementName(String)` replaces the method `setTableName(String)`; `setPrimaryKeyElementName(String)` replaces `setPrimaryKeyFieldName(String)`; and so on.

XMLPlatform

XMLPlatform is a subclass of SDKPlatform that implements the methods required to support sequence numbers: `buildSelectSequenceCall()` and `buildUpdateSequenceCall()`. These methods build and return the XMLDataCalls that allow TopLink to use sequence numbers that are maintained in XML documents.

The root element name for these XML documents and the names of the elements used to hold the sequence name and sequence counter can be set by your application via the XMLFileLogin.

XMLFileLogin

XMLFileLogin is a subclass of SDKLogin that allows for the configuration of the XMLFileAccessor and XMLPlatform. The XMLFileLogin is used to configure the following settings:

- The base directory name for the XML files. This is the directory under which the root element name sub-directories are located. For more information on file name resolution, see "XMLFileAccessor" on page 5-32. The default is the current working directory.

```
login.setBaseDirectoryName("C:\Employee Database");
```

- The file name extension for the XML files. The default is ".xml".

```
login.setFileExtension(".xml");
```

- Whether directories for the XML files should be created as needed. The default is false.

```
login.setCreatesDirectoriesAsNeeded(true);
```

- Sequence number settings.

```
login.setSequenceRootElementName("sequence");
login.setSequenceNameElementName("name");
login.setSequenceCounterElementName("count");
```

XMLSchemaManager

XMLSchemaManager is a subclass of SDKSchemaManager that provides support for building the XML-based sequences required by your TopLink DatabaseSession. After you have built your TopLink Project and used it to create a DatabaseSession, you can log in and create the required sequences with the XMLSchemaManager.

```
DatabaseSession session = project.createDatabaseSession();
session.login();
SchemaManager manager = new XMLSchemaManager(session);
manager.createSequences();
```

XMLAccessor

XMLAccessor is an interface that extends the `oracle.toplink.internal.databaseaccess.Accessor` interface and, by default, is used by the XMLCalls to access the appropriate Stream for a given XML document.

You can provide your own implementation of this interface if you want TopLink to read and write your XML documents from and to something other than the native

file system. If for example, your XML documents are accessed via a messaging service such the Java Messaging Service (JMS), you could develop an implementation of XMLAccessor that would translate the method calls into the appropriate invocations of the messaging service, whether it be reading, writing, or deleting an XML document. Once you have developed your custom Accessor, you could configure an XMLLogin to use it.

```
XMLLogin login = new XMLLogin();
login.setAccessorClass(XMLJMSAccessor.class);
login.setUserName("user");
login.setPassword("password");
// etc.
```

XMLTranslator

XMLTranslator is an interface that is used by the XMLCalls to convert XML documents to TopLink DatabaseRows and vice versa. Each XMLCall has its own XMLTranslator. By default, this is an instance of DefaultXMLTranslator. This can be overridden by your own custom implementation of XMLTranslator. The protocol defined by XMLTranslator is very simple:

- The method `read(java.io.Reader)` takes a Reader that streams over an XML document, converts that document into a DatabaseRow, and returns that DatabaseRow.
- The method `write(java.io.Writer, DatabaseRow)` takes a DatabaseRow and converts it into an XML document and writes that document out on the Writer.

DefaultXMLTranslator

The default XMLTranslator used by the XMLCalls, DefaultXMLTranslator, performs a fairly straightforward set of translations to convert a DatabaseRow into an XML document and vice versa. Here is a summary of the translations, expressed in terms of converting a DatabaseRow into an XML document (the reverse set of translations are just the opposite):

- All the fields in the DatabaseRow must have the same table name; otherwise, an XMLDataStoreException is thrown. This table name is used for the root element name of the XML document.

```
<?xml version="1.0"?>
<employee>
<!-- field values will go here -->
</employee>
```

- Each field in the DatabaseRow becomes an XML element. The field name becomes the element name, while the field value becomes the element content.

```
<?xml version="1.0"?>
<employee>
    <id>1</id>
    <firstName>Grace</firstName>
    <lastName>Hopper</lastName>
</employee>
```

- Any field in the DatabaseRow with a value of null becomes an empty XML element with an attribute named null whose value is "true".

```
<managedEmployees null="true"/>
```

- If the value of a field in the DatabaseRow is an SDKFieldValue, the elements of the SDKFieldValue are converted into nested XML elements. If the elements of the SDKFieldValue are also DatabaseRows, these are translated recursively, using the same set of translations.

SDKAggregateObjectMapping

```
<?xml version="1.0"?>
<employee>
    <id>1</id>
    <firstName>Grace</firstName>
    <lastName>Hopper</lastName>
    <period>
        <employmentPeriod>
           <startDate>1943-01-01</startDate>
           <endDate>1992-01-01</endDate>
        </employmentPeriod>
    </period>
</employee>
```

SDKDirectCollectionMapping

```
<?xml version="1.0"?>
<employee>
    <id>1</id>
    <firstName>Grace</firstName>
    <lastName>Hopper</lastName>
    <responsibilities>
        <responsibility>find bugs</responsibility>
        <responsibility>develop compilers</responsibility>
    </responsibilities>
</employee>
```

SDKObjectCollectionMapping

```
<?xml version="1.0"?>
<employee>
    <id>1</id>
    <firstName>Grace</firstName>
    <lastName>Hopper</lastName>
    <phoneNumbers>
        <phone>
           <areaCode>888</areaCode>
           <number>555-1212</number>
           <type>work</type>
        </phone>
        <phone>
           <areaCode>800</areaCode>
           <number>555-1212</number>
           <type>home</type>
        </phone>
    </phoneNumbers>
</employee>
```

The DefaultXMLTranslator delegates the actual translating to two other classes:

- DatabaseRowToXMLTranslator
- XMLToDatabaseRowTranslator

DatabaseRowToXMLTranslator The DatabaseRowToXMLTranslator performs the translations previously mentioned, building and XML document from a DatabaseRow and writing it onto a Stream.

XMLToDatabaseRowTranslator The XMLToDatabaseRowTranslator performs the reverse of the translations previously described, reading the XML document from a Stream and building a DatabaseRow. To accomplish this conversion, the XMLToDatabaseRowTranslator uses the Xerces XML parser to parse the XML document.

Note: The XMLToDatabaseRowTranslator is in a separate package from the other XML classes (*oracle.toplink.xml.xerces*) so that it can be loaded dynamically if necessary. If your application, or any third-party class library used by your application, uses an XML parser, you may need to configure TopLink to use a custom class loader.

This custom class loader allows TopLink to use the specific version of Xerces shipped with TopLink, without interfering with any other parser your application may be using.

To activate this custom class loader, configure TopLink to use the classes in the JAR files shipped with TopLink instead of the classes found by the system class loader. Call a static method defined in DatabaseLogin and pass in the names of the two JAR files shipped with TopLink that contain the Xerces parser and the TopLink classes that use the Xerces parser.

```
DatabaseLogin.setXMLParserJARFileNames(new String[] {"xerces.jar",
"toplinksdkxerces.jar"});
```

XML Zip File Extension

The XML Zip file extension is an enhancement to the XML implementation of the SDK. This extension adds the flexibility of maintaining the XML data store in a group of archive files rather than in the directory/file structure of the standard XML data store. The format is very similar to the standard XML data store however, the directories, which essentially represent tables, are now replaced with archive files. The contents of the archive files are the XML documents.

Using the Zip file extension

Using the XML Zip file extension is straightforward. In most situations it only requires the addition of one line of code.

Configure XMLLogin Typically, you need only configure your XMLLogin to use a different Accessor.

```
XMLLogin login = new XMLLogin();
login.setAccessorClass(XMLZipFileAccessor.class);
```

Configure direct file access with Zip File extension

There is one other difference that you may encounter if you are configuring XMLCalls to access files directly. To access an XML document within an archive file, the call needs to know both the archive file location and the name of the XML document entry within the archive. Therefore, the `setFileName()` message sent to an XMLCall needs to include both the archive file and the XML document entry name.

```
XMLReadCall call = new XMLReadCall();
call.setFileName("C:/Employee DataStore/employee.zip", "1.xml");
```

Implementation details

Only two classes make up the Zip file extension, these classes are in the package `oracle.toplink.xml.zip`.

- XMLZipFileAccessor
- XMLZipFileStreamPolicy

XMLZipFileAccessor The XMLZipFileAccessor extends the XMLFileAccessor. It essentially performs the same function as its standard XML package counterpart with the exception of using the XMLZipFileStreamPolicy rather than the XMLFileStreamPolicy used in the XMLFileAccessor. There is no added functionality – it simply subclasses XMLFileAccessor to provide substitutability in the XMLLogin.

XMLZipFileStreamPolicy This class is the most significant change from the standard XML package. It handles the XML archive files. It returns streams for reading and writing from individual archive entries. It does not provide any additional functionality over its standard XML package counterpart, the XMLFileStreamPolicy. It transparently provides the same functionality, whilst handling the added complication of getting read/write streams from within an archive file.

6

Performance Optimization

Designing for peak efficiency ensures that your TopLink application is fast, smooth, and accurate. This chapter discusses how to optimize TopLink-enabled applications. It discusses

- Basic performance optimization
- TopLink writing optimization features
- Schema optimization

Basic performance optimization

Performance consideration should be factored into every part of the development cycle. This means that you should be aware of performance issues in your design and implementation. This does not mean, however, that you should try to optimize performance in the first iteration. Optimizations that complicate the design or implementation should be left until the final iteration of your application. However, you should plan for these performance optimizations from your first iteration to make it easier to integrate them later.

The single most important aspect of performance optimization is *knowing what to optimize*. To improve the performance of your application, you must fully understand exactly what areas of your application have performance problems. You must also fully understand the causes of performance problems.

TopLink provides a diverse set of features to optimize performance. Most of these features can be turned on or off in the descriptors and/or database session and result in a global system performance improvement, without any changes to application code.

When optimizing the performance of your application, you should first check to see if a TopLink feature can solve the optimization problem. If no such feature is

present then you should consider more complex optimizations, such as those provided in the later sections of this chapter.

TopLink reading optimization features

Certain read and write operations can be optimized through TopLink. The following two key concepts are used to optimize reading:

- changing how much data is read from the database
- changing the way the data is queried from the database

Table 6–1 lists the read optimization features provided with TopLink.

Table 6–1 Read optimization features

Feature	Effect on performance
Unit of Work	Tracks object changes within the unit of work. Only register objects that will change to minimize the amount of tracking required.
Object indirection	"Value holders" are used to stand in for real domain objects to avoid reading them until they are accessed. The usage of value holders is strongly recommended as they provide a major performance benefit.
Weak identity map	Client-side caching of objects read from database. The client-side cache holds only objects referenced by the application. Avoids database calls by reading objects from cache. Efficient use of memory. The benefit of caching with the weak identity map may not be as great as the soft cache weak identity map, but it uses less memory.
Soft cache weak identity map	Client-side caching of objects read from database. The client-side cache holds only objects referenced by the application and releases objects not referenced by the application when memory becomes low. Avoids database calls by reading objects from cache. Efficient use of memory. Gives the benefit of caching, but does not cause memory problems.
Full identity map	Client side caching of objects read from the database. This permits database calls to be avoided if the object has already been read in. **Caution**: Ensure that the cache size does not grow too large, as this may cause severe performance problems.
Cache identity map	Client side cache that will always use only a fixed amount of memory. Gives the benefit of caching, but does not cause memory problems.

Table 6–1 Read optimization features (Cont.)

Feature	Effect on performance
No identity map	Cache lookup can be avoided completely for objects that do not need to be cached.
Batch reading and joining	Both of these features can be used to dramatically reduce the number of database accesses that are required to perform a read query. Reduces database access by batching many queries into a single query that reads more data.
Partial object reading	Allows reading of a subset of a result set of the object's attributes. Reduces the amount of data that needs to be read from the database to improve performance.
Report query	Similar to partial object reading, but returns only the data instead of the objects. Gives the same performance benefit as partial object reading. The report query also supports complex reporting functions such as aggregation functions and "group by". Complex results can be computed on the database instead of reading the objects into the application and computing the result in memory.

Reading Case 1: Displaying names in a list - optimized through partial object reading and report query

An application often asks the user to choose a particular element from a list. The list displays only a subset of the information contained in the objects, and therefore it is wasteful to query all of the information for all of the objects from the database. It is possible to query only the information required to display in the list, and then, when the user chooses one, read only that object from the database.

TopLink has two features, partial object reading and report query, that allow the performance of these types of operations to be optimized.

Partial object reading

Partial object reading is a query designed to extract only the required information from a selected record in a database, rather than all of the information the record contains.

When using partial object reading, the object is not fully populated, so it cannot be cached. Consequently, the object cannot be edited. Because the primary key is required to re-query the object (so it can be edited for example), and because TopLink does not automatically include the primary key information in a partially populated object, the primary key must be explicitly specified as a partial attribute.

Example 6–1 No optimization

```
/* Read all the employees from the database, ask the user to choose one and
return it. This must read in all the information for all of the employees.*/
List list;

// Fetch data from database and add to list box.
Vector employees = (Vector) session.readAllObjects(Employee.class);
list.addAll(employees);

// Display list box.
....

// Get selected employee from list.
Employee selectedEmployee = (Employee) list.getSelectedItem();

return selectedEmployee;
```

Example 6–2 Optimization through partial object reading

```
/* Read all the employees from the database, ask the user to choose one and
return it. This uses partial object reading to read just the last name of the
employees. Note that TopLink does not automatically include the primary key of
the object. If this is needed to select the object for a query, it must be
specified as a partial attribute so that it can be included. In this way, the
object can easily be read for editing. */
List list;
// Fetch data from database and add to list box.
ReadAllQuery query = new ReadAllQuery(Employee.class);
query.addPartialAttribute("lastName");
// add this if the primary key is required for re-querying the object
query.addPartialAttribute("id");
/* TopLink does not automatically include the primary key of the object. If this
is needed to select the object for a query, it must be specified as a partial
attribute so that it can be included.*/
query.addPartialAttribute("id");
query.dontMaintainCache();
Vector employees = (Vector) session.executeQuery(query);
list.addAll(employees);

// Display list box.
....
// Get selected employee from list.
Employee selectedEmployee =
(Employee)session.readObject(list.getSelectedItem());
return selectedEmployee;
```

Note: If `query.dontMaintainCache()` is not included in this example, a query exception is thrown.

Example 6–3 Optimization through report query

```
/* Read all the employees from the database, ask the user to choose one and
return it. This uses the report query to read just the last name of the
employees. It then uses the primary key stored in the report query result to
read the real object.*/
List list;
// Fetch data from database and add to list box.
ExpressionBuilder builder = new ExpressionBuilder();
ReportQuery query = new ReportQuery (Employee.class, builder);
query.addAttribute("lastName");
query.retrievePrimaryKeys();
Vector reportRows = (Vector) session.executeQuery(query);
list.addAll(reportRows);

// Display list box.
....

// Get selected employee from list.
Employee selectedEmployee = (Employee)
((ReportResult)list.getSelectedItem()).readObject;

return selectedEmployee;
```

Conclusion

Although the differences between the two examples are slight, there is a substantial performance improvement by using partial objects and report query.

In the example called "No optimization" on page 6-4, all of the full employee objects are created even though only the employee's last name is displayed in the list. All of the data that makes up an employee object must be read.

In the example called "Optimization through partial object reading" on page 6-4, partial object reading is used to read only the last name (and the primary key, if specified) of the employees. Read employee objects are still created, but only the last name (and primary key) is set. The other employee attributes are left as null or as their constructor defaults. This reduces the amount of data read from the database.

In this example, the report query is used to read only the last name of the employees. This reduces the amount of data read from the database and avoids instantiating any employee instances.

Specifying fewer partial attributes and querying larger objects improves the overall performance gain of these optimizations.

Reading Case 2: Batch reading objects

The amount of data read by your application affects performance, but how that data is read also affects performance.

Reading a collection of rows from the database is significantly faster than reading each row individually. The most common performance problem is reading a collection of objects that have a one-to-one reference to another object. If this is done without optimizing how the objects are read, N + 1 database calls are required. That is, one read operation is required to read in all of the source rows, and one call for each target row is required in the one-to-one relationship.

The next three examples show a two-phase query that reads the addresses of a set of employees individually, and then reads them using TopLink's query optimization features. The optimized read accesses the database only twice, so it is significantly faster.

Example 6–4 No optimization

```
/*Read all the employees, and collect their address' cities. This takes N + 1
queries if not optimized. */

// Read all of the employees from the database. This requires 1 SQL call.
Vector employees = session.readAllObjects(Employee.class,new
ExpressionBuilder().get("lastName").equal("Smith"));

//SQL: Select * from Employee where l_name = 'Smith'

// Iterate over employees and get their addresses.
// This requires N SQL calls.
Enumeration enum = employees.elements();
Vector cities = new Vector();
while(enum.hasMoreElements()) Employee employee = (Employee) enum.nextElement();
cities.addElement(employee.getAddress().getCity());

//SQL: Select * from Address where address_id = 123, etc }
```

Example 6–5 Optimization through joining

```
/* Read all the employees, and collect their address' cities. Although the code
is almost identical because joining optimization is used it only takes 1 query.
*/

// Read all of the employees from the database, using joining. This requires 1 SQL
call.
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.setSelectionCriteria(new
ExpressionBuilder().get("lastName").equal("Smith"));
query.addJoinedAttribute("address");
Vector employees = session.executeQuery(query);

// SQL: Select E.*, A.* from Employee E, Address A where E.l_name = 'Smith' and
E.address_id = A.address_id Iterate over employees and get their addresses. The
previous SQL already read all of the addresses so no SQL is required.
Enumeration enum = employees.elements();
Vector cities = new Vector();
while (enum.hasMoreElements()) {
Employee employee = (Employee) enum.nextElement();
    cities.addElement(employee.getAddress().getCity());
```

Example 6–6 Optimization through batch reading

```
/* Read all the employees, and collect their address' cities. Although the code
is almost identical because batch reading optimization is used it only takes 2
queries. */

// Read all of the employees from the database, using batch reading. This
requires 1 SQL call, note that only the employees are read.
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.setSelectionCriteria(new
ExpressionBuilder().get("lastName").equal("Smith"));
query.addBatchReadAttribute("address");
Vector employees = (Vector)session.executeQuery(query);

// SQL: Select * from Employee where l_name = 'Smith'

// Iterate over employees and get their addresses.
// The first address accessed will cause all of the addresses to be read in a
single SQL call.
Enumeration enum = employees.elements();
Vector cities = new Vector();
```

```
while (enum.hasMoreElements()) {
    Employee employee = (Employee) enum.nextElement();
    cities.addElement(employee.getAddress()
    .getCity());
    // SQL: Select distinct A.* from Employee E, Address A where E.l_name =
    'Smith' and E.address_id = A.address_i
}
```

Conclusion

By using TopLink query optimization, a number of queries are reduced to a single query. This leads to much greater performance.

It may seem that because joining requires only a single query that batch reading would never be required. The advantage of batch reading is that it allows for delayed loading through value holders and has much better performance where the target objects are shared. For example, if all of the employees lived at the same address, batch reading would read much less data than joining, because batch reading uses a SQL DISTINCT to filter duplicate data. Batch reading is also supported for one-to-many relationships where joining is supported only for one-to-one relationships.

Although this technique is very efficient, it should only be used when all of the desired objects (such as addresses) are required. Otherwise the resources spent reading all of the objects could hurt performance.

Reading Case 3: Using complex custom SQL queries

TopLink provides a high-level query mechanism. This query mechanism is powerful, but currently does not support everything possible through raw SQL. If you have a complex query required by your application, and the query must be done optimally, the best solution in many cases is to use raw SQL.

Reading Case 4: Viewing objects

Some parts of an application may require information from a variety of objects rather than from just one object. This can be very difficult to implement and very performance intensive. In such situations, it may be advantageous to define a new read-only object to encapsulate this information and map it to a view on the database. Set the object to be read-only by using the `addDefaultReadOnlyClass()` API in the `oracle.toplink.sessions.Project` class.

Example 6–7 No optimization

```
/* Gather the information to report on an employee and return the summary of the
information. In this situation a hashtable is used to hold the report
information. Notice that this reads a lot of objects from the database, but uses
very little of the information contained in the objects. This may take 5 queries
and read in a large number of objects.*/


public Hashtable reportOnEmployee(String employeeName)
    {
    Vector projects, associations;
    Hashtable report = new Hashtable();
    // Retrieve employee from database.
    Employee employee = session.readObject(Employee.class, new
    ExpressionBuilder.get("lastName").equal(employeeName));
    // Get all of the projects affiliated with the employee.
    projects = session.readAllObjects(Project.class, "SELECT P.* FROM PROJECT P,
    EMPLOYEE E WHERE P.MEMBER_ID = E.EMP_ID AND E.L_NAME = " + employeeName);
    // Get all of the associations affiliated with the employee.associations
    =session.readAllObjects(Association.class, "SELECT A.* FROM ASSOC A,
    EMPLOYEE E WHERE A.MEMBER_ID = E.EMP_ID AND E.L_NAME = " + employeeName);
}

report.put("firstName", employee.getFirstName());
report.put("lastName", employee.getLastName());
report.put("manager", employee.getManager());
report.put("city", employee.getAddress().getCity());
report.put("projects", projects);
report.put("associations", associations);
return report;}
```

Example 6–8 Optimization through view object

```
CREATE VIEW NAMED EMPLOYEE_VIEW AS (SELECT F_NAME = E.F_NAME, L_NAME = E.L_
NAME,EMP_ID = E.EMP_ID, MANAGER_NAME = E.NAME, CITY = A.CITY, NAME = E.NAME
FROM EMPLOYEE E, EMPLOYEE M, ADDRESS A
WHERE E.MANAGER_ID = M.EMP_ID
AND E.ADDRESS_ID = A.ADDRESS_ID)
```

Then, define a descriptor for the `EmployeeReport` class:

- Define the descriptor as normal; however, set `tableName` to be `EMPLOYEE_VIEW`.
- Map only the attributes required for the report; in the case of `numberOfProjects` and associations, a transformation mapping can be used to get the required data.

Now, the report can be queried from the database like any other TopLink-enabled object.

Example 6–9 With optimization

```
/* Return the report for the employee.*/
public EmployeeReport reportOnEmployee(String employeeName)
{
    EmployeeReport report;
    report = (EmployeeReport) session.readObject(EmployeeReport.class, new
    ExpressionBuilder.get("lastName").equal
    (employeeName));
    return report;}
```

TopLink writing optimization features

Table 6–2 lists the write optimization features provided with TopLink.

Table 6–2 Write optimization features

Feature	Effect on performance
Unit of Work	Minimal update of object changes on commit of the unit of work. Improves performance by updating only the changed fields and objects. Tracks object changes within the unit of work. Minimizes the amount of tracking required (which can be expensive) by registering only those objects that will change. **Note**: The unit of work supports marking classes as read-only, which allows the unit of work to avoid tracking changes of objects that will not be changed.
Parameterized SQL	The session or an individual query can be configured to use a prepared statement and cache the statement, thus avoiding the SQL prepare call on subsequent executions of the query. Performance improves in situations when the same SQL statement is executed many times.
Batch writing	Supported in both JDK 1.1 and JDK 1.2. Allows for all of the insert, update, and delete commands from a transaction to be grouped into a single database call. Performance improves dramatically because the number of calls to the database is reduced.
Sequence number preallocation	Sequence numbers are cached (pre-allocated) on the client side to dramatically improve insert performance.

Table 6–2 Write optimization features (Cont.)

Feature	Effect on performance
Does exist alternatives	"Does exist" call on write object can be avoided in certain situations by checking the cache for "does exist" or assuming existence.

Writing Case 1: Batch writes

TopLink also provides several write optimization features. The most common write performance problem is a batch job that inserts a large volume of data into the database.

Consider a batch job that requires to load a large amount of data from one database and migrate the data into another. Assume that the objects are simple employee objects that use generated sequence numbers as their primary key, and have an address that also uses a sequence number. The batch job requires to load 10,000 employees from the first database and insert them into the target database.

First lets approach the problem naively and have the batch job read all of the employees from the source database, and then acquire a unit of work from the target database, register all of the objects and commit the unit of work.

Example 6–10 No optimization

```
/* Read all the employees, acquire a unit of work and register them. */

// Read all of the employees from the database. This requires 1 SQL call, but
will be very memory intensive as 10,000 objects will be read.
Vector employees = sourceSession.readAllObjects(Employee.class);

//SQL: Select * from Employee

// Acquire a unit of work and register the employees.
UnitOfWork uow = targetSession.acquireUnitOfWork();
uow.registerAllObjects(employees);
uow.commit();

//SQL: Begin transaction
//SQL: Update Sequence set count = count + 1 where name = 'EMP'
//SQL: Select count from Sequence
//SQL: ... repeat this 10,000 times + 10,000 times for the addresses ...
//SQL: Commit transaction
//SQL: Begin transaction
//SQL: Insert into Adresss (...) values (...)
```

```
//SQL: ... repeat this 10,000 times
//SQL: Insert into Employee (...) values (...)
//SQL: ... repeat this 10,000 times
//SQL: Commit transaction}
```

This batch job would have extremely poor performance and would cause 60,000 SQL executions. It also reads huge amounts of data into memory that can cause memory performance issues. There are a number of TopLink optimization that can be used to optimize this batch job.

Batching and cursoring

The first performance problem is that loading from the source database may cause memory problems. To optimize the problem, a cursored stream should be used to read the employees from the source database. Also, a cache identity map should be used in both the source and target databases, not a full identity map (a weak identity map could be used in JDK 1.2).

The cursor should be streamed in groups of 100 using the `releasePrevious()` method after each read. Each batch of 100 employees should be registered in a new unit of work and committed. Although this does not change the amount of SQL executed, it does fix the memory problems. You should be able to notice a memory problem in a batch job through noticing the performance degrading over time and possible disk swapping occurring.

Sequence number pre-allocation

SQL select calls are more expensive than SQL modify calls, so the biggest performance gain is in reducing any select being issued. In this example, selects are used for the sequence numbers. Using sequence number pre-allocation dramatically improves the performance.

In TopLink, the sequence pre-allocation size can be configured on the login; it defaults to 50. In the non-optimized example, we used a pre-allocation size of 1 to demonstrate this point. Because batches of 100 are used, a sequence pre-allocation size of 100 should also be used. Because both employees and address use sequence number, we can get even better pre-allocation by having them share the same sequence. In this case, we set the pre-allocation size to 200. This optimization reduces the number of SQL execution from 60,000 to 20,200.

Batch writing

TopLink supports batch writing on batch compliant databases in JDK 1.1 and through batch compliant JDBC 2.0 drivers in JDK 1.2. Batch writing allows for a group of SQL statements to be batched together into a single statement and sent to the database as a single database execution. This reduces the communication time between the application and the server and can lead to huge performance increases.

Batch writing can be enabled on the login through the `useBatchWriting()` method. In our example, each batch of 100 employees can be batched into a single SQL execution. This reduces the number of SQL execution from 20,200 to 300.

Parameterized SQL

TopLink supports parameterized SQL and prepared statement caching. Using parameterized SQL can improve write performance by avoiding the prepare cost of a SQL execution through reusing the same prepared statement for multiple executions.

Batch writing and parameterized SQL cannot be used together, because batch writing does not use individual statements. The performance benefits of batch writing are much greater than parameterized SQL; therefore, if batch writing is supported by your database, it is strongly suggested that you use batch writing and not use parameterized SQL.

Parameterized SQL avoids only the prepare part of the SQL execution, not the execute; therefore, it normally does not give a huge performance gain. However, if your database does not support batch writing, parameterized SQL can improve performance. In this example, the number of SQL executions is still 20,200, but the number of SQL prepares is reduced to 4.

Multi-processing

Multiple processes and even multiple machines can be used to split the batch job into several smaller jobs.

Splitting the batch job across ten threads leads to performance increases. In this case, the read from the cursored stream could be synchronized and parallel units of work could be used on a single machine.

Even if the machine has only a single processor, this can lead to a performance increase. During any SQL execution the thread must wait for a response from the server, but in this waiting time the other threads can be processing.

The final optimized example does not show multi-processing as normally the other features are enough to improve the performance.

Example 6–11 Fully optimized

```
/* Read each batch of employees, acquire a unit of work and register them. */
targetSession.getLogin().useBatchWriting();
targetSession.getLogin().setSequencePreallocationSize(200);

// Read all of the employees from the database, into a stream. This requires 1
SQL call, but none of the rows will be fetched.
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Employee.class);
query.useCursoredStream();
CursoredStream stream;
stream = (CursoredStream) sourceSession.executeQuery(query);
//SQL: Select * from Employee. Process each batch
    while (! stream.atEnd()) {
    Vector employees = stream.read(100);
// Acquire a unit of work to register the employees
        UnitOfWork uow = targetSession.acquireUnitOfWork();
        uow.registerAllObjects(employees);
        uow.commit();
    }
    //SQL: Begin transaction
    //SQL: Update Sequence set count = count + 200 where name = 'SEQ'
    //SQL: Select count from Sequence where name = 'SEQ'
    //SQL: Commit transaction
    //SQL: Begin transaction
    //BEGIN BATCH SQL: Insert into Address (...) values (...)
        //... repeat this 100 times
        //Insert into Employee (...) values (...)
        //... repeat this 100 times
    //END BATCH SQL:
    //SQL: Commit transactionJava optimization
```

In most client-server database applications, most of the performance problems come from the communications between the client and the server. This means that optimizing Java code is normally not as important as optimizing database interactions. However, you should still try to write clean, optimized Java code, since very poorly optimized Java code does affect the performance of your application.

Optimization check list

The following is a general checklist to keep in mind when developing Java applications.

- Do not make code more complicated than necessary.
- Write encapsulated code so that complex behavior can be easily optimized within the encapsulation.
- Use instance or static variables to cache the results of expensive computations.
- Use hash tables for large collections that are looked up by key.
- Always provide default sizes to vectors and hash tables if only a few elements will be added to them.
- Postpone executing expensive tasks until absolutely necessary.
- Make use of multi-tasking to perform background jobs.
- When performing a lot of `String` manipulations, use a `StringBuffer` instead of the `+` operator for appending `Strings`.
- Consider lazy initialization in cases where the value's initialization in the constructor is normally not required.
- If using RMI or CORBA, avoid fine-grain remote message sends.

Schema optimization

When designing your database schema and object model, optimization is very important. The key element to remember in the design of your object model and database schema is to avoid complexity. The most common object-relational performance problem is when the database schema is derived directly from a complex object model. This normally produces an over-normalized database schema that can be slow and difficult to query.

Although it is best to design the object model and database schema together, there should not be a direct one-to-one mapping between the two.

Schema Case 1: Aggregation of two tables into one

A common schema optimization technique is to de normalize two tables into one. This can improve read and write performance by requiring only one database operation instead of two.

This technique is demonstrated through analyzing the ACME Member Location Tracking System.

Table 6–3 Original schema

Elements	Details			
Title	ACME Member Location Tracking System			
Classes	Member, Address			
Tables	MEMBER, ADDRESS			
Relationships	Source	Instance Variable	Mapping	Target
	Member	address	one-to-one	Address

Table 6–4 Optimized schema

Elements	**Details**			
Classes	Member, Address			
Tables	MEMBER			
Relationships	Source	Instance Variable	Mapping	Target
	Member	address	aggregate	Address

Domain

In the ACME Member Location Tracking System, employees and addresses are always looked up together.

Problem Querying a member based on address information requires an expensive database join. Reading a member and its address requires two read statements. Writing a member requires two write statements. This unnecessarily adds complexity to the system and results in poor performance.

Solution Since members are always read and written with their address information, considerable performance can be gained through combining the MEMBER and

ADDRESS tables into a single table, and changing the one-to-one relationship to an aggregate relationship.

This allows all of the information to be read in a single operation, and doubles the speed of updates and inserts as only one row from one table is modified.

Schema Case 2: Splitting one table into many

This example demonstrates how a table schema can be further normalized to provide performance optimization.

Frequently, relational schemas can stuff too much data into a particular table. The table may contain a large number of columns, but only a small subset of those may be frequently used.

By splitting the large table into two or even several smaller tables, the amount of data traffic can be significantly reduced, improving the overall performance of the system.

Table 6–5 Original schema

Elements	Details			
Title	ACME Employee Workflow System			
Classes	Employee, Address, PhoneNumber, EmailAddress, JobClassification, Project			
Tables	EMPLOYEE, PROJECT, PROJ_EMP			
Relationships	Source	Instance Variable	Mapping	Target
	Employee	address	aggregate	Address
	Employee	phoneNumber	aggregate	EmailAddress
	Employee	emailAddress	aggregate	EmailAddress
	Employee	job	aggregate	JobClassification
	Employee	projects	many-to-many	Project

Table 6–6 Optimized schema

Elements	Details
Classes	Employee, Address, PhoneNumber, EmailAddress, JobClassification, Project

Table 6–6 Optimized schema (Cont.)

Elements	Details			
Tables	EMPLOYEE, ADDRESS, PHONE, EMAIL, JOB, PROJECT, PROJ_EMP			
Relationships	Source	Instance Variable	Mapping	Target
	Employee	address	one-to-one	Address
	Employee	phoneNumber	one-to-one	EmailAddress
	Employee	emailAddress	one-to-one	EmailAddress
	Employee	job	one-to-one	JobClassification
	Employee	projects	many-to-many	Project

Domain

This system is responsible for assigning employees to projects within an organization. The most-common operation is to read a set of employees and projects, assign some employees to different projects, and update the employees. Occasionally the employee's address or job classification is used to determine which project would be the best placement for the employee.

Problem When a large volume of employees is read from the database at one time, their aggregate parts must also be read. Because of this, the system suffers from a general read performance problem. The only solution is to reduce the amount of data traffic to and from the server.

Solution In this system, normalize the EMPLOYEE table into the EMPLOYEE, ADDRESS, PHONE, EMAIL, and JOB tables.

Since normally only the employee information is read, the amount of data transferred from the database to the client is reduced by splitting the table. This improves your read performance by reducing the amount of data traffic by 25%.

Schema Case 3: Collapsed hierarchy

When models are designed in an object-oriented design and then transformed into a relational model, a common mistake is to make a large hierarchy of tables on the database. This makes it necessary to perform a large number of joins and makes querying difficult. Normally it is a good idea to collapse some of the levels in your inheritance hierarchy into a single table.

Table 6–7 Original schema

Elements	Details
Title	ACME Sales Force System
Classes	Tables
Person	PERSON
Employee	PERSON, EMPLOYEE
SalesRep	PERSON, EMPLOYEE, REP
Staff	PERSON, EMPLOYEE, STAFF
Client	PERSON, CLIENT
Contact	PERSON, CONTACT

Table 6–8 Optimized schema

Elements	Details
Classes	Tables
Person	<none>
Employee	EMPLOYEE
SalesRep	EMPLOYEE
Staff	EMPLOYEE
Client	CLIENT
Contact	CLIENT

Domain

In this system, the clients of the company are assigned to its sales force representatives. The managers track which sales representatives are under them.

Problem The system suffers from over-complexity, which hinders the development and performance of the system. Large expensive joins are required to do almost anything, making every database operation expensive.

Solution By collapsing the three-level table hierarchy into one, the complexity of the system is reduced. All of the expensive joins in the system are eliminated and

simplified queries allow read performance to be further optimized leading to greatly improved system performance.

Schema Case 4: Choosing one out of many

A common situation is for an object to have a collection of other objects where only one of the other objects in the collection is commonly used. In this situation, it is desirable to add an instance variable just for this special object. This way, the important object can be accessed and used without requiring the instantiation of all of the other objects in the collection.

Table 6–9 Original schema

Elements	Details			
Title	ACME Shipping Package Location Tracking System			
Classes	Package, Location			
Tables	PACKAGE, LOCATION			
Relationships	Source	Instance Variable	Mapping	Target
	Package	locations	one-to-many	Location

Table 6–10 Optimized schema

Elements	Details			
Classes	Package, Location			
Tables	PACKAGE, LOCATION			
Relationships	Source	Instance Variable	Mapping	Target
	Package	locations	one-to-many	Location
	Package	currentLocation	one-to-one	Location

Domain

This system is used by an international shipping company, which wants to be able to track the location of its packages as they travel from their source to their destination. When a package is moved from one location to another, a location is created in real-time on the database. The application normally receives a request for the current location of a particular package and displays this for the user.

Problem A package could accumulate many locations as it travels to its destination, so reading all of these locations from the database is expensive.

Solution By adding a specific instance variable for just the current location and a one-to-one mapping for the instance variable, the current location can be accessed without reading in all of the other locations. This drastically improves the performance of the system.

7

Mapping Implementation

Mapping enables you to relate objects in your application to data in a database. This chapter discusses how you can use Java code to implement mappings in TopLink-based applications. It discusses

- Direct mappings
- Relationship mappings
- Object relational mappings

For detailed descriptions of these mappings, see the *Oracle9iAS TopLink Mapping Workbench Reference Guide*.

Direct mappings

Direct mappings define how a persistent object refers to objects that do not have descriptors, such as the JDK classes and primitives. There are several types of direct mappings, including

- Direct-to-field mappings
- Type conversion mappings
- Object type mappings
- Serialized object mappings
- Transformation mappings

Direct-to-field mappings

In Java, use the `DirectToFieldMapping` class to create direct-to-field mappings. The mapping requires you to set the following:

- Attribute mapped, set by sending the `setAttributeName()` message
- Field to store the value of the attribute, set by sending the `setFieldName()` message

The optional `setGetMethodName()` and `setSetMethodName()` messages allow TopLink to access the attribute through user-defined methods rather than directly through the attribute. You do not have to define the accessors when using Java 2.

The `Descriptor` class provides the `addDirectMapping()` method that can create a new `DirectToFieldMapping`, set the attribute and field name parameters, and register the mapping with the descriptor.

Example 7–1 Creating a direct-to-field mapping in Java and registering it with the descriptor.

```
// Create a new mapping and register it with the descriptor.
DirectToFieldMapping mapping = new DirectToFieldMapping();
mapping.setAttributeName("city");
mapping.setFieldName("CITY");
descriptor.addMapping(mapping);
```

Example 7–2 Creating a mapping that uses method access.

This mapping example assumes persistent class has `getCity()` and `setCity()` methods defined.

```
// Create a new mapping and register it with the descriptor.
DirectToFieldMapping mapping = new DirectToFieldMapping();
mapping.setAttributeName("city");
mapping.setFieldName("CITY");
mapping.setGetMethodName("getCity");
mapping.setSetMethodName("setCity");
descriptor.addMapping(mapping);
```

Example 7–3 Using the two overloaded versions of the descriptor's addDirectMapping() method

```
// Alternate method which does the same thing.
descriptor1.addDirectMapping("city", "CITY");
descriptor2.addDirectMapping("city", "getCity",
```

Reference

Table 7–1 summarizes the most common public methods for direct-to-field mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for direct-to-field mapping, see the TopLink JavaDocs.

Table 7–1 Elements for direct-to-field mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Field to be mapped *	not applicable	`setFieldName(String name)`

* Required property

Type conversion mappings

Create type conversion mappings with the `TypeConversionMapping` class. The following elements are required for a type conversion mapping:

- The attribute mapped, set by sending the `setAttributeName()` message
- The field to store the value of the attribute, set by `setFieldName()` message
- The Java type stored in the attribute, set by sending the `setAttributeClassification()` message
- The database type to be written, set by sending the `setFieldClassification()` message

The optional `setGetMethodName()` and `setSetMethodName()` messages allow TopLink to access the attribute through user-defined methods rather than directly through the attribute. You do not have to define the accessors when using Java 2.

Example 7–4 Creating a type conversion mapping and registering it with the descriptor

```
// Create a new mapping and register it with the descriptor.
TypeConversionMapping typeConversion = new TypeConversionMapping();
typeConversion.setFieldName("J_DAY");
typeConversion.setAttributeName("joiningDate");
typeConversion.setFieldClassification(java.sql.Date.class);
```

```
typeConversion.setAttributeClassification(java.util.Date.class);
descriptor.addMapping(typeConversion);
```

Reference

Table 7–2 summarizes the most common public methods for type conversion mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for type conversion mapping, see the TopLink JavaDocs.

Table 7–2 Elements for type conversion mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Field to be mapped *	not applicable	`setFieldName(String name)`
Attribute/field classification *	attribute type	`setAttributeClassification(Class aClass)` `setFieldClassification(Class aClass)`

* Required

Object type mappings

Object type mappings are instances of the `ObjectTypeMapping` class. The following elements are required for an object type mapping:

- The attribute mapped, set by sending the `setAttributeName()` message
- The field to store the value of the attribute, set by `setFieldName()` message
- A set of values and their conversions, added by sending the `addConversionValue()` message

The optional `setGetMethodName()` and `setSetMethodName()` messages allow TopLink to access the attribute through user-defined methods rather than directly. You do not have to define the accessors when using Java.

The following two methods are useful in a legacy environment, or you want to change the values of the fields:

- `addToAttributeOnlyConversionValue (Object fieldValue, Object attributeValue)` – adds conversion value only for the field value to attribute value. This is a one-way mapping from the field to the attribute. This can be used if multiple database values are to mapped to the same object value. When written to the database, the value entered by `addConversionValue(Object fieldValue, Object attributeValue)` is used and therefore the original values in the database change.
- `setDefaultAttributeValue (Object defaultAttributeValue)` – the default value can be used if the database can store values in addition to those that have been mapped. Any value retrieved from database that is not mapped is substituted for the default value. When writing to the database, the value entered by `addConversionValue(Object fieldValue, Object attributeValue)` is used and therefore the original values in the database change.

Example 7–5 Creating an object type mapping and registering it with the descriptor

```
// Create a new mapping and register it with the descriptor.
ObjectTypeMapping typeMapping = new ObjectTypeMapping();
typeMapping.setAttributeName("gender");
typeMapping.setFieldName("GENDER");
typeMapping.addConversionValue("M", "Male");
typeMapping.addConversionValue("F", "Female");
typeMapping.setNullValue("F");
descriptor.addMapping(typeMapping);
```

Reference

Table 7–3 summarizes the most common public methods for object type mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for object type mapping, see the TopLink JavaDocs.

Table 7–3 Elements for object type mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Field to be mapped *	not applicable	`setFieldName(String name)`

* Required property

Serialized object mappings

Serialized object mappings are instances of the `SerializedObjectMapping` class. The following elements are *required* for a serialized object mapping:

- The attribute mapped, set by sending the `setAttributeName()` message
- The field that stores the value of the attribute, set by the `setFieldName()` message

The optional `setGetMethodName()` and `setSetMethodName()` messages allow TopLink to access the attribute through user-defined methods rather than directly through the attribute. You do not have to define accessors when using Java 2.

Example 7–6 Creating a serialized object mapping and registering it with the descriptor.

```
// Create a new mapping and register it with the descriptor.
SerializedObjectMapping serializedMapping = new SerializedObjectMapping();
serializedMapping.setAttributeName("jobDescription");
serializedMapping.setFieldName("JOB_DESC");
descriptor.addMapping(serializedMapping);
```

Reference

Table 7–4 summarizes the most common public methods for serialized object mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for serialized object mapping, see the TopLink JavaDocs.

Table 7–4 Elements for serialized object mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Field to be mapped *	not applicable	`setFieldName(String name)`

* Required property

Transformation mappings

Transformation mappings enable you to create specialized translations between how a value is represented in Java and in the database. Use transformation mappings only when mapping multiple fields into a single attribute. Transformation mapping is often appropriate when values from multiple fields are used to create an object.

> **Note:** Because of the complexity of transformation mappings, it may be easier in some cases to perform the transformation with get/set methods of a direct-to-field mapping.

Implementing transformation mappings in Java

Transformation mappings are instances of the `TransformationMapping` class. The following elements are typically required for a mapping:

- The attribute mapped, set by sending the `setAttributeName()` message; not required for write-only mappings
- The method to be invoked that sets the value of the attribute from information in the database row; set by sending the `setAttributeTransformation()` message that expects one or two parameters, a `DatabaseRow` and optionally a `Session`
- A set of methods associated to fields in the database, where the value for each field is the result of invoking the associated method; associations are made by sending the `addFieldTransformation()` message, passing along the database field name and the method name

The optional `setGetMethodName()` and `setSetMethodName()` messages allow TopLink to access the attribute through user-defined methods rather than directly.

Example 7–7 Creating a transformation mapping and registering it with the descriptor.

This particular example provides custom support for two of the fields number of fields can be mapped using this approach.

```
// Create a new mapping and register it with the descriptor.
TransformationMapping transformation1 = new TransformationMapping();
transformation1.setAttributeName ("dateAndTimeOfBirth");
transformation1.setAttributeTransformation ("buildDateAndTime");
transformation1.addFieldTransformation("B_DAY", "getDateOfBirth");
transformation1.addFieldTransformation("B_TIME", "getTimeOfBirth");
descriptor.addMapping(transformation1);
// Define attribute transformation method to read from the database row
public java.util.Date buildDateAndTime(DatabaseRow row) {
    java.sql.Date sqlDateOfBirth = (java.sql.Date)row.get("B_DAY");
    java.sql.Time timeOfBirth = (java.sql.Time)row.get("B_TIME");
    java.util.Date utilDateOfBirth = new java.util.Date(
        sqlDateOfBirth.getYear(),
        sqlDateOfBirth.getMonth(),
        sqlDateOfBirth.getDate(),
        timeOfBirth.getHours(),
        timeOfBirth.getMinutes(),
        timeOfBirth.getSeconds());
    return utilDateOfBirth;
}

// Define a field transformation method to write to the database
public java.sql.Time getTimeOfBirth()
{
    return new java.sql.Time this.dateAndTimeOfBirth.getHours(),
    this.dateAndTimeOfBirth.getMinutes(), this.dateAndTimeOfBirth.getSeconds());
}

// Define a field transformation method to write to the database
public java.sql.Date getDateOfBirth()
{
    return new java.sql.DateOfBirth this.dateAndTimeOfBirth.getYear(),
    this.dateAndTimeOfBirth.getMonth(), this.dateAndTimeOfBirth.getDate());
}
```

Example 7–8 Creating a transformation mapping using indirection

```
// Create a new mapping and register it with the descriptor.
TransformationMapping transformation2 = new
transformation2.setAttributeName("designation");
```

```
transformation2.setGetMethodName ("getDesignationHolder");
transformation2.setSetMethodName ("setDesignationHolder");
transformation2.setAttributeTransformation ("getRankFromRow");
transformation2.addFieldTransformation("RANK", "getRankFromObject");
transformation2.useIndirection();
descriptor.addMapping(transformation2);

//Define an attribute transformation method to read from database row.
public String getRankFromRow()
{
    Integer value = new Integer(((Number)row.get("RANK")).intValue());
    String rank = null;
    if (value.intValue() == 1) {
        rank = "Executive";
    }
    if (value.intValue() == 2) {
        rank = "Non-Executive";
    }
    return rank;
}
//Define a field transformation method to write to the database.
public Integer getRankFromObject()
{
    Integer rank = null;

    if (getDesignation().equals("Executive"))
    rank = new Integer(1);
    if (getDesignation().equals("Non-Executive"))
    rank = new Integer(2);
    return rank;
}

//Provide accessor methods for the indirection.
private ValueHolderInterface designation;
public ValueHolderInterface getDesignationHolder()
{
    return designation;
}
public void setDesignationHolder(ValueHolderInterface value)
{
    designation = value;
}
```

Reference

Table 7–5 summarizes the most common public methods for transformation mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for transformation mapping, see the TopLink JavaDocs.

Table 7–5 Elements for transformation mapping

Element	Default	Method Names
Attribute to be mapped	not applicable	`setAttributeName(String name)`
Transformations	not applicable	`addFieldTransformation(String fieldName, String methodName)` `setAttributeTransformation(String methodName)`

Relationship mappings

Relational mappings define how persistent objects reference other persistent objects. These mappings include all of the following:

- Aggregate object mappings
- One-to-one mappings
- Variable one-to-one mappings
- Direct collection mappings
- Direct map mappings
- One-to-many mappings
- Many-to-many mappings

Aggregate object mappings

Aggregate object mappings are instances of the `AggregateObjectMapping` class. This mapping is associated to an attribute in each of the parent classes. The following elements are required for an aggregate object mapping to be viable:

- The attribute mapped, set by sending the `setAttributeName()` message
- The target (child) class, set by sending the `setReferenceClass()` message

The optional `setGetMethodName()` and `setSetMethodName()` messages allow TopLink to access the attribute through user-defined methods rather than directly.

By default the mapping allows null references to its target class, so it does not create an instance of the target object. To prevent a parent from having a null reference, send the `dontAllowNull()` message, which results in an instance of the child with its attributes set to null.

The following modifications to the target (child) class descriptor are required:

- an indication that all information will come from its parent's row(s); this is accomplished by sending the `descriptorIsAggregate()` message to the descriptor
- no table or primary key information

Example 7–9 Creating an aggregate object mapping for the Employee source class and registering it with the descriptor.

```
// Create a new mapping and register it with the source descriptor.
AggregateObjectMapping aggregateMapping = new AggregateObjectMapping();
aggregateMapping.setAttributeName("employPeriod");
aggregateMapping.setReferenceClass(Period.class);
descriptor.addMapping(aggregateMapping);
```

Example 7–10 Creating the descriptor of the Period aggregate target class.

The aggregate target descriptor does not need a mapping to its parent, and does not need any table or primary key information.

```
// Create a descriptor for the aggregate class. The table name and primary key
are not specified in the aggregate descriptor.
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(Period.class);
descriptor.descriptorIsAggregate();
```

```
// Define the attribute mappings or relationship mappings.
descriptor.addDirectMapping("startDate", "START_DATE");
descriptor.addDirectMapping("endDate", "END_DATE");
return descriptor;
```

Example 7–11 Creating an aggregate object mapping for the Project which is another source class that contains a Period.

The field names must be translated in the Project descriptor as shown in Table 7–1. No changes need to be made to the `Period` class descriptor to implement this second parent.

```
// Create a new mapping and register it with the parent descriptor.
AggregateObjectMapping aggregateMapping = new AggregateObjectMapping();
aggregateMapping.setAttributeName("projectPeriod");
aggregateMapping.setReferenceClass(Period.class);
aggregateMapping.addFieldNameTranslation("S_DATE", "START_DATE");
aggregateMapping.addFieldNameTranslation("E_DATE", "END_DATE");
descriptor.addMapping(aggregateMapping);
```

Reference

Table 7–6 summarizes the most common public methods for aggregate object mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for aggregate object mapping, see the TopLink JavaDocs.

Table 7–6 Elements for aggregate object mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Set parent class *	not applicable	`setReferenceClass(Class aClass)`

* Required property

One-to-one mappings

One-to-one mappings, are instances of the `OneToOneMapping()` class, that require the following:

- The attribute mapped, set by sending the `setAttributeName()` message
- The reference class, set by sending the `setReferenceClass()` message
- Foreign key information, normally specified by sending the `setForeignKeyFieldName()` message and passing the foreign key field from the source table that references the primary key of the target table.

> **Note:** If the target primary key is composite, send the `addForeignKeyFieldName()` message for each of the foreign fields and target primary key fields that make up the relationship.

If the mapping has a bi-directional relationship where the two classes in the relationship reference each other with one-to-one mappings, then to properly set up the foreign key information:

- One mapping must send the `setForeignKeyFieldName()` message.
- The other must send the `setTargetForeignKeyFieldName()` message.

It is also possible to set up composite foreign key information by sending the `addForeignKeyFieldName()` and `addTargetForeignKeyFieldName()` messages.

> **Note:** Indirection is enabled by default, requiring that the attribute be a `ValueHolderInterface`.

Example 7–12 Creating a simple one-to-one mapping and registering it with the descriptor.

```
// Create a new mapping and register it with the descriptor.
OneToOneMapping oneToOneMapping = new OneToOneMapping();
oneToOneMapping.setAttributeName("address");
oneToOneMapping.setReferenceClass(Address.class);
oneToOneMapping.setForeignKeyFieldName("ADDRESS_ID");
descriptor.addMapping(oneToOneMapping);
```

Example 7–13 Implementing a bidirectional mapping between two classes that reference each other.

The foreign key is stored in the Policy's table referencing the composite primary key of the Carrier.

```
// In the Policy class, which will hold the foreign key, create the mapping
which references the Carrier class.
OneToOneMapping carrierMapping = new OneToOneMapping();
carrierMapping.setAttributeName("carrier");
carrierMapping.setReferenceClass(Carrier.class);
carrierMapping.addForeignKeyFieldName("INSURED_ID", "CARRIER_ID");
carrierMapping.addForeignKeyFieldName("INSURED_TYPE", "TYPE");
descriptor.addMapping(carrierMapping);. . .
// In the Carrier class, create the mapping which references the Policy class.
OneToOneMapping policyMapping = new OneToOneMapping();
policyMapping.setAttributeName("masterPolicy");
policyMapping.setReferenceClass(Policy.class);
policyMapping.addTargetForeignKeyFieldName("INSURED_ID", "CARRIER_ID");
policyMapping.addTargetForeignKeyFieldName("INSURED_TYPE", "TYPE");
descriptor.addMapping(policyMapping);
```

Reference

Table 7–7 summarizes the most common public methods for one-to-one mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for one-to-one mapping, see the TopLink JavaDocs.

Table 7–7 Elements for one-to-one mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Foreign key indicator (at least one of these) *	not applicable	`addForeignKeyFieldName( String foreignKeyFieldName, String targetKeyFieldName)`
Referenced class *	not applicable	`setReferenceClass(Class referenceClass)`

* Required property

Variable one-to-one mappings

Variable one-to-one mappings are instances of the `VariableOneToOneMapping()` class. The following elements are required for a variable one-to-one mapping:

- The attribute mapped, set by sending the `setAttributeName()` message
- The reference class, set by sending the `setReferenceClass()` message
- Foreign key and target query key information, normally specified by sending the `setForeignQueryKeyName()` message and passing the source foreign key field name and the target abstract query key name on the interface descriptor.

> **Note:** If the target implementor descriptors' primary keys are composite, send the `addForeignQueryKeyName()` message for each of the foreign key fields and target query keys that make up the relationship.

If the mapping uses a class indicator field:

- A type indicator field must be specified.
- The class indicator values are specified on the mapping so that mapping can determine the class of object to create.

> **Note:** Indirection is enabled by default, requiring that the attribute be a `ValueHolderInterface`.

Example 7–14 Defining a variable one-to-one mapping using a class indicator field.

```
VariableOneToOneMapping variableOneToOneMapping = new VariableOneToOneMapping();
variableOneToOneMapping.setAttributeName("contact");
variableOneToOneMapping.setReferenceClass (Contact.class);
variableOneToOneMapping.setForeignQueryKeyName ("C_ID", "id");
variableOneToOneMapping.setTypeFieldName("TYPE");
variableOneToOneMapping.addClassIndicator(Email.class, "Email");
variableOneToOneMapping.addClassIndicator(Phone.class, "Phone");
variableOneToOneMapping.dontUseIndirection();
variableOneToOneMapping.privateOwnedRelationship();
```

Example 7–15 Defining a variable one-to-one mapping using a unique primary key

```
VariableOneToOneMapping variableOneToOneMapping = new VariableOneToOneMapping();
variableOneToOneMapping.setAttributeName("contact");
variableOneToOneMapping.setReferenceClass (Contact.class);
variableOneToOneMapping.setForeignQueryKeyName ("C_ID", "id");
variableOneToOneMapping.dontUseIndirection();
variableOneToOneMapping.privateOwnedRelationship();
```

Reference

Table 7–8 summarizes the most common public methods for variable one-to-one mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for variable one-to-one mapping, see the TopLink JavaDocs.

Table 7–8 Elements for variable one-to-one mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Field to query key mapping *	none	`setForeignQueryKeyName(String foreignKeyName, String abstractQueryKeyName)` `addForeignQueryKeyName(String foreignKeyName, String abstractQueryKeyName)`
Referenced class *	not applicable	`setReferenceClass(Class referenceClass)`

Read only	read / write	`readWrite()`
		`readOnly()`
		`setIsReadOnly(boolean isReadOnly)`

* Required property

Direct collection mappings

Direct collection mappings are instances of the `DirectCollectionMapping` class. The following elements are required for a direct collection mapping:

- The attribute mapped, set by sending the `setAttributeName()` message
- The database table that holds the values to be stored in the collection, set by sending the `setReferenceTableName()` message

- The field in the reference table from which the values are read and placed into the collection; this is called the direct field and is set by sending the `setDirectFieldName()` message
- Foreign key information, which is specified by sending the `setReferenceKeyFieldName()` message and passing the name of the field that is a foreign reference to the primary key of the source object.

> **Note:** If the target primary key is composite, send the `addReferenceKeyFieldName()` message for each of the fields that make up the key.

Example 7–16 Creating a simple direct collection mapping.

```
DirectCollectionMapping directCollectionMapping = new DirectCollectionMapping();
directCollectionMapping.setAttributeName ("responsibilitiesList");
directCollectionMapping.setReferenceTableName ("RESPONS");
directCollectionMapping.setDirectFieldName("DESCRIP");
directCollectionMapping.setReferenceKeyFieldName ("EMP_ID");
directCollectionMapping.useCollectionClass (Vector.class); // the default
descriptor.addMapping(directCollectionMapping);
```

Reference

Table 7–9 summarizes the most common public methods for direct collection mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for direct collection mapping, see the TopLink JavaDocs.

Table 7–9 Elements for direct collection mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Primary key information *	not applicable	`addReferenceKeyFieldName(String referenceKey, String sourceKey)`

* Required property

Table 7–9 Elements for direct collection mapping (Cont.)

Element	Default	Method Names
Reference table information *	not applicable	`setDirectFieldName(String fieldName)` `setReferenceTableName(String tableName)`

Indirection	use indirection	`useBasicIndirection()`
		`useTransparentCollection()`
		`dontUseIndirection()`

Method access	direct access	`setGetMethodName(String name)` `setSetMethodName(String name)`
Collection Type	Vector	`useCollectionClass(Class)`

* Required property

Aggregate collections

Aggregate collection mappings are used to represent the aggregate relationship between a single-source object and a collection of target objects. Unlike the TopLink one-to-many mappings, in which there should be a one-to-one back reference mapping from the target objects to the source object, there is no back reference required for the aggregate collection mappings because the foreign key relationship is resolved by the aggregation.

To implement an aggregate collection mapping:

- The descriptor of the target class must declare itself to be an aggregate collection object. Unlike the aggregate object mapping, in which the target descriptor does not have a specific table to associate with, there must be a target table for the target object.
- The descriptor of the source class must add an aggregate collection mapping that specifies the target class.

When to use aggregate collections

Although similar in behavior to 1-many mappings, an aggregate collection is not a replacement for 1-many mappings, and great care should be exercised when choosing an aggregate collection over a one-to-many mapping. aggregate collections should only be used in situations where the target collections are reasonable in size and having a one-to-one mapping from the target to the source is difficult.

Consider using a one-to-many relationship rather than an aggregate collection, because one-to-many relationships offer better performance and are more robust and scalable.

In addition, aggregate collections are privately owned by the source of the relationship and should not be shared or referenced by other objects.

Aggregate collections and inheritance

Aggregate collection descriptors can make use of inheritance. The subclasses must also be declared as aggregate collection. The subclasses can have their own mapped tables, or share the table with their parent class.

In a Java Vector, the owner references its parts. In a relational database, the parts reference their owners. Relational databases use this implementation to make querying more efficient.

Java implementation

Aggregate collection mappings are instances of the `AggregateCollectionMapping` class. The following elements are required for an aggregate collection mapping:

- The attribute mapped, set by sending the `setAttributeName()` message
- The reference class, set by sending the `setReferenceClass()` message.
- Foreign key information, which is specified by sending the `addTargetForeignKeyFieldName()` message and passing the field name of the target foreign key and the source the primary key in the source table.

Notes:

- If the source primary key is composite, send the `addTargetForeignKeyFieldName()` message to each of the fields that make up the key.
- Indirection is enabled by default for an aggregate collection mapping, requiring that the attribute implement `ValueHolderInterface`.

Example 7–17 Creating a simple aggregate collection mapping and registering it with the descriptor

```
// In the PolicyHolder class, create the mapping which references the Phone
class
AggregateCollectionMapping phonesMapping = new AggregateCollectionMapping();
phonesMapping.setAttributeName("phones");
phonesMapping.setGetMethodName("getPhones");
phonesMapping.setSetMethodName("setPhones");
phonesMapping.setReferenceClass("Phone.class");
phonesMapping.dontUseIndirection();
phonesMapping.privateOwnedRelationship;
phonesMapping.addTargetForeignKeyFieldName ("INS_PHONE.HOLDER_
SSN","HOLDER.SSN");
descriptor.addMapping(phonesMapping);
```

Reference

Table 7–10 summarizes the most common public methods for aggregate collection mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for aggregate collection mapping, see the TopLink JavaDocs.

Table 7–10 Properties for aggregate collection mapping

Property	Default	Method Names
Attribute to be mapped	not applicable	`setAttributeName(String name)`
Foreign key indicator	not applicable	`addTargetForeignKeyFieldName (String targetForeignKeyFieldName, String sourceKeyFieldName)`
Referenced class	not applicable	`setReferenceClass(Class aClass)`
Indirection	use indirection	`useBasicIndirection()` `useTransparentCollection()` `dontUseIndirection()`
Method access	direct access	`setGetMethodName(String name)` `setSetMethodName(String name)`
Privately owned relationship	privately owned	`privateOwnedRelationship()`

Table 7–10 Properties for aggregate collection mapping (Cont.)

Property	Default	Method Names
Collection type	Vector	`useCollectionClass(Class)`

Direct map mappings

Direct map mappings store instances that implement `java.util.Map`. Unlike one-to-manys or many to manys, the keys and values of the map in this type of mapping are Java objects that do not have descriptors. The object type stored in the key and the value of direct map are Java primitive wrapper types such as String objects.

Support for primitive data types such as int is not provided since Java maps only hold objects.

Direct map mappings are instances of the `DirectMapMapping` class. The following elements are required for a direct map mapping:

- The attribute mapped, set by sending the `setAttributeName()` message
- The database table that holds the keys and values to be stored in the map, set by sending the `setReferenceTableName()` message
- The field in the reference table from which the keys are read and placed into the map; this is called the direct key field and is set by sending the `setDirectKeyFieldName()` message
- Foreign key information, which is specified by sending the `setReferenceKeyFieldName()` message and passing the name of the field that is a foreign reference to the primary key of the source object

  Note: If the target primary key is composite, you must send the `addReferenceKeyFieldName()` message for each of the fields that make up the key

- The field in the reference table from which the values are read and placed into the map; this is called the direct field and is set by sending the `setDirectFieldName()` message
- The Java type of key in map from which the keys are converted from keys read from the database placed into the map; this is set by sending the `setKeyClass()` message

- The Java type of value in map from which the values are converted from values read from the database placed into the map; this is set by sending the `setValueClass()` message

Example 7–18 Creating a simple direct map mapping.

```
DirectMapMapping directMapMapping = new DirectMapMapping();
directMapMapping.setAttributeName("cities");
directMapMapping.setReferenceTableName("CITY_TEMP");
directMapMapping.setReferenceKeyFieldName("RECORD_ID");
directMapMapping.setDirectKeyFieldName("CITY");
directMapMapping.setDirectFieldName("TEMPERATURE");
directMapMapping.setKeyClass(String.class);
directMapMapping.setValueClass(Integer.class);

descriptor.addMapping(directMapMapping);
```

Reference

Table 7–11 summarizes the most common public methods for direct map mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for direct map mapping, see the TopLink JavaDocs.

Table 7–11 Elements for direct map mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Primary key information *	not applicable	`addReferenceKeyFieldName(String referenceKey, String sourceKey)`
Reference table information *	not applicable	`setDirectFieldName(String fieldName)` `setReferenceTableName(String tableName)`

Indirection	use indirection	`useBasicIndirection()`
		`dontUseIndirection()`
		`useTransparentMap()`

Method access	direct access	`setGetMethodName(String name)` `setSetMethodName(String name)`

* Required property

One-to-many mappings

One-to-many mappings are instances of the `OneToManyMapping` class. The following elements are required for a one-to-many mapping:

- The attribute mapped, set by sending the `setAttributeName()` message
- The reference class, set by sending the `setReferenceClass()` message
- Foreign key information, which is specified by sending the `setTargetForeignKeyFieldName()` message and passing a field in the target object's associated table that refers to the primary key in the owning object's table.

> **Note:** If the target primary key is composite, send the `addTargetForeignKeyFieldName()` message for each of the fields that make up the key.

- A one-to-one mapping in the target class back to the source class; (see "Elements for direct-to-field mapping" on page 7-3 for more information)

> **Note:** Indirection is enabled by default for a one-to-many mapping, requiring that the attribute implement `ValueHolderInterface`.

Example 7–19 Creating a simple one-to-many mapping and registering it with the descriptor

```
// In the Employee class, create the mapping which references the Phone class.
oneToManyMapping = new OneToManyMapping();
oneToManyMapping.setAttributeName("phoneNumbers");
oneToManyMapping.setReferenceClass(PhoneNumber.class);
oneToManyMapping.setTargetForeignKeyFieldName("EMPID");
descriptor.addMapping(oneToManyMapping);
. . .
// In the Phone class, which will hold the foreign key, create the mapping which
references the Employee class.
OneToOneMapping oneToOneMapping = new OneToOneMapping();
oneToOneMapping.setAttributeName("owner");
oneToOneMapping.setReferenceClass(Employee.class);
oneToOneMapping.setForeignKeyFieldName("EMPID");
descriptor.addMapping(oneToOneMapping);
```

Reference

Table 7–12 summarizes the most common public methods for one-to-many mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for one-to-many mapping, see the TopLink JavaDocs.

Table 7–12 Elements for one-to-many mapping

Property	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Foreign key indicator *	not applicable	`addTargetForeignKeyFieldName(String targetForeignKeyFieldName, String sourceKeyFieldName)`
Referenced class *	not applicable	`setReferenceClass(Class aClass)`

Indirection	use indirection	`useBasicIndirection()`
		`useTransparentCollection()`
		`dontUseIndirection()`

Method access	direct access	`setGetMethodName(String name)` `setSetMethodName(String name)`
Privately owned relationship	independent	`privateOwnedRelationship()` `setIsPrivateOwned(Boolean isPrivateOwned)`
Collection Type	Vector	`useCollectionClass(Class)`

* Required property

Many-to-many mappings

Many-to-many mappings are instances of the `ManyToManyMapping` class. The following elements are required for a many-to-many mapping:

- The attribute mapped, set by sending the `setAttributeName()` message
- The reference class, set by sending the `setReferenceClass()` message
- The relation table, set by sending the `setRelationTableName()` message

- Foreign key information, which is specified by sending the `setSourceRelationKeyFieldName()` and `setTargetRelationKeyFieldName()` messages; if the source or target primary keys are composite, send the `addSourceRelationKeyFieldName()` or `addTargetRelationKeyFieldName()` messages

Example 7–20 Code that creates a simple many-to-many mapping

```
// In the Employee class, create the mapping which references the Project class.
ManyToManyMapping manyToManyMapping = new ManyToManyMapping();
manyToManyMapping.setAttributeName("projects");
manyToManyMapping.setReferenceClass(Project.class);
manyToManyMapping.setRelationTableName("PROJ_EMP");
manyToManyMapping.setSourceRelationKeyFieldName ("EMPID");
manyToManyMapping.setTargetRelationKeyFieldName ("PROJID");
descriptor.addMapping(manyToManyMapping);
```

Reference

Table 7–13 summarizes the most common public methods for many-to-many mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for many-to-many mapping, see the TopLink JavaDocs.

Table 7–13 Elements for many-to-many mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Referenced class *	not applicable	`setReferenceClass(Class aClass)`
Relation table *	not applicable	`setRelationTableName(String tableName)`
Relation keys *	not applicable	`addSourceRelationKeyFieldName(String sourceRelationKey, String sourceKey)` `addTargetRelationKeyFieldName(String targetRelationKey, String targetKey)`

* Required property

Table 7–13 Elements for many-to-many mapping (Cont.)

Element	Default	Method Names

Indirection	use indirection	`useBasicIndirection()`
		`useTransparentCollection()`
		`dontUseIndirection()`

Privately owned relationship	independent	`privateOwnedRelationship()`
Method access	direct access	`setGetMethodName(String name)` `setSetMethodName(String name)`
Collection Type	Vector	`useCollectionClass(Class)`

* Required property

Object relational mappings

Relational mappings define how persistent objects reference other persistent objects. Object relational mappings allow for an object model to be persisted into an object-relational data-model. TopLink does not directly support these mappings – they must be defined in code through amendment methods.

TopLink enables you to leverage the following object relational mapping types:

- Array mappings
- Object array mappings
- Structure mappings
- Reference mappings
- Nested table mappings

Array mappings

In an object-relational data-model, structures can contain *arrays* (collections of other data types). These arrays can contain primitive data types or collections of other structures. TopLink stores the arrays with their parent structure in the same table.

TopLink supports arrays of primitive data through the `ArrayMapping`. This is similar to `DirectCollectionMapping` – it represents a collection of primitives in Java. However, the `ArrayMapping` does not require an additional table to store the values in the collection.

Implementing array mappings in Java

Array mappings are instances of the `ArrayMapping` class. You must associate this mapping to an attribute in the parent class. TopLink requires the following elements for an array mapping:

- Attribute mapped, set by sending the `setAttributeName( )` message.
- Field mapped, set by sending the `setFieldName()` message.
- Name of the array, set by sending the `setStructureName()` message.

Example 7–21 Creating an array mapping for the Employee source class and registering it with the descriptor

```
// Create a new mapping and register it with the source descriptor.
ArrayMapping arrayMapping = new ArrayMapping();
arrayMapping.setAttributeName("responsibilities");
arrayMapping.setStructureName("Responsibilities_t");
arrayMapping.setFieldName("RESPONSIBILITIES");
descriptor.addMapping(arrayMapping);
```

Reference

Table 7–14 summarizes the most common public methods for array mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for array mapping, see the TopLink JavaDocs.

Table 7–14 Elements for array mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Set parent class *	not applicable	`setReferenceClass(Class referenceClass)`
User-defined data type *	not applicable	`setStructureName(String structureName)`
Field to be mapped *	not applicable	`setFieldName(String fieldName)`

* Required property

Table 7–14 Elements for array mapping (Cont.)

Element	Default	Method Names
Method access	direct access	`setGetMethodName(String name)`
		`setSetMethodName(String name)`
Read only	read / write	`readWrite()`
		`readOnly()`
		`setIsReadOnly(boolean readOnly)`

* Required property

Object array mappings

In an object-relational data-model, object arrays allow for an array of object types or structures to be embedded into a single column in a database table or an object table.

Implementing object array mappings in Java

Object array mappings are instances of the `ObjectArrayMapping` class. You must associate this mapping to an attribute in the parent class. TopLink requires the following elements for an array mapping:

- Attribute mapped, set by sending the `setAttributeName()` message.
- Field mapped, set by sending the `setFieldName()` message.
- Name of the array, set by sending the `setStructureName ()` message.

Use the optional `setGetMethodName()` and `setSetMethodName()` messages to access the attribute through user-defined methods rather than directly.

Example 7–22 Creating an object array mapping for the Insurance source class and registering it with the descriptor.

```
// Create a new mapping and register it with the source descriptor.
ObjectArrayMapping phonesMapping = new ObjectArrayMapping();
phonesMapping.setAttributeName("phones");
phonesMapping.setGetMethodName("getPhones");
phonesMapping.setSetMethodName("setPhones");
phonesMapping.setStructureName("PHONELIST_TYPE");
phonesMapping.setReferenceClass(Phone.class);
phonesMapping.setFieldName("PHONES");
descriptor.addMapping(phonesMapping);
```

Reference

Table 7–15 summarizes the most common public methods for object array mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for object array mapping, see the TopLink JavaDocs.

Table 7–15 Elements for object array mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Set parent class *	not applicable	`setReferenceClass(Class referenceClass)`
User-defined data type *	not applicable	`setStructureName(String structureName)`
Field to be mapped *	not applicable	`setFieldName(String fieldName)`
Method access	direct access	`setGetMethodName(String name)` `setSetMethodName(String name)`

Read only	read / write	`readWrite()`
		`readOnly()`
		`setIsReadOnly(boolean readOnly)`

* Required property

Structure mappings

In an object-relational data-model, structures are user defined data-types or object-types. This is similar to a Java class – it denies attributes or fields in which each attribute is either:

- a primitive data type
- another structure
- reference to another structure

TopLink maps each structure to a Java class defined in your object model and defines a descriptor for each class. A `StructureMapping` maps nested structures, similar to an `AggregateObjectMapping`. However, the structure mapping supports

null values and shared aggregates without requiring additional settings (because of the object-relational support of the database).

Implementing structure mappings in Java

Structure mappings are instances of the `StructureMapping` class. You must associate this mapping to an attribute in each of the parent classes. TopLink requires the following elements for an array mapping:

- Attribute mapped, set by sending the `setAttributeName()` message.
- Field mapped, set by sending the `setFieldName()` message.
- Target (child) class, set by sending the `setReferenceClass()` message.

Use the optional `setGetMethodName()` and `setSetMethodName()` messages to access the attribute through user-defined methods rather than directly.

You must make the following changes to the target (child) class descriptor:

- Send the `descriptorIsAggregate()` message to indicate it is not a root level.
- Remove table or primary key information.

Example 7–23 Creating a structure mapping for the Employee source class and registering it with the descriptor

```
// Create a new mapping and register it with the source descriptor.
StructureMapping structureMapping = new StructureMapping();
structureMapping.setAttributeName("address");
structureMapping.setReferenceClass(Address.class);
structureMapping.setFieldName("address");
descriptor.addMapping(structureMapping);
```

Example 7–24 Creating the descriptor of the Address aggregate target class.

The aggregate target descriptor does not need a mapping to its parent, or any table or primary key information.

```
// Create a descriptor for the aggregate class. The table name and primary key
are not specified in the aggregate descriptor.
ObjectRelationalDescriptor descriptor = new ObjectRelationalDescriptor ();
descriptor.setJavaClass(Address.class);
descriptor.setStructureName("ADDRESS_T");
descriptor.descriptorIsAggregate();

// Define the field ordering
descriptor.addFieldOrdering("STREET");
```

```
descriptor.addFieldOrdering("CITY");
...

// Define the attribute mappings or relationship mappings.
...
```

Reference

Table 7–16 summarizes the most common public methods for structure mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for structure mapping, see the TopLink JavaDocs.

Table 7–16 Elements for structure mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Set parent class *	not applicable	`setReferenceClass(Class aClass)`
Field to be mapped *	not applicable	`setFieldName(String fieldName)`
Method access	direct access	`setGetMethodName(String name)` `setSetMethodName(String name)`

Read only	read / write	`readWrite()`
		`readOnly()`
		`setIsReadOnly(boolean readOnly)`

* Required property

Reference mappings

In an object-relational data-model, structures reference each other through *refs* – not through foreign keys (as in a traditional data-model). *Refs* are based on the target structure's `ObjectID`.

TopLink supports refs through the `ReferenceMapping`. They represent an object reference in Java, similar to a `OneToOneMapping`. However, the reference mapping does not require foreign key information.

Implementing reference mappings in Java

Reference mappings are instances of the `ReferenceMapping` class. You must associate this mapping to an attribute in the source class. TopLink requires the following elements for a reference mapping:

- Attribute mapped, set by sending the `setAttributeName( )` message.
- Field mapped, set by sending the `setFieldName( )` message.
- Target class, set by sending the `setRefrenceClass ( )` message.

Use the optional `setGetMethodName( )` and `setSetMethodName( )` messages to access the attribute through user-defined methods rather than directly.

Example 7–25 Creating a reference mapping for the Employee source class and registering it with the descriptor.

```
// Create a new mapping and register it with the source descriptor.
ReferenceMapping referenceMapping = new ReferenceMapping();
referenceMapping.setAttributeName("manager");
referenceMapping.setReferenceClass(Employee.class);
referenceMapping.setFieldName("MANAGER");
descriptor.addMapping(refrenceMapping);
```

Reference

Table 7–17 summarizes the most common public methods for reference mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for reference mapping, see the TopLink JavaDocs.

Table 7–17 Elements for reference mapping

Property	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Set parent class *	not applicable	`setReferenceClass(Class aClass)`
Field to be mapped *	not applicable	`setFieldName(String fieldName)`

* Required property

Table 7–17 Elements for reference mapping (Cont.)

Property	Default	Method Names
Method access	direct access	`setGetMethodName(String name)` `setSetMethodName(String name)`
Indirection	use indirection	`useBasicIndirection()` `dontUseIndirection()`

Privately owned relationship	independent	`independentRelationship()`
		`privateOwnedRelationship()`
		`setIsPrivateOwned(boolean isPrivateOwned)`

Read only	read / write	`readWrite()`
		`readOnly()`
		`setIsReadOnly(boolean readOnly)`

* Required property

Nested table mappings

Nested table types model an unordered set of elements. These elements may be built-in or user-defined types. Nested tables typically represent a one-to-many or many-to-many relationship of references to another independent structure. They support querying and joining better than Varrays that are inlined to the parent table.

TopLink supports nested table through the `NestedTableMapping`. They represent a collection of object references in Java, similar to a `OneToManyMapping` or `ManyToManyMapping`. However, the nested table mapping does not require foreign key information (like a one-to-many mapping) or the relational table (like a many-to-many mapping).

Implementing nested table mappings in Java

Nested table mappings are instances of the `NestedTableMapping` class. This mapping is associated to an attribute in the parent class. The following elements are required for a nested table mapping to be viable:

- The attribute mapped, set by sending the `setAttributeName()` message
- The field mapped, set by sending the `setFieldName()` message

- The name of the array structure, set by sending the `setStructureName()` message

Use the optional `setGetMethodName()` and `setSetMethodName()` messages to allow TopLink to access the attribute through user-defined methods rather than directly.

Example 7–26 Creating a nested table mapping for the Insurance source class and registering it with the descriptor.

```
// Create a new mapping and register it with the source descriptor.
NestedTableMapping policiesMapping = new NestedTableMapping();
policiesMapping.setAttributeName("policies");
policiesMapping.setGetMethodName("getPolicies");
policiesMapping.setSetMethodName("setPolicies");
policiesMapping.setReferenceClass(Policy.class);
policiesMapping.dontUseIndirection();
policiesMapping.setStructureName("POLICIES_TYPE");
policiesMapping.setFieldName("POLICIES");
policiesMapping.privateOwnedRelationship();
policiesMapping.setSelectionSQLString("select p.* from policyHolders ph,
table(ph.policies) t, policies p where ph.ssn=#SSN and ref(p) = value(t)");
descriptor.addMapping(policiesMapping);
```

Reference

Table 7–18 summarizes the most common public methods for nested table mapping:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for nested table mapping, see the TopLink JavaDocs.

Table 7–18 Elements for nested table mapping

Element	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Set parent class *	not applicable	`setReferenceClass(Class referenceClass)`

* Required property

Table 7–18 Elements for nested table mapping (Cont.)

Element	Default	Method Names
User-defined data type *	not applicable	`setStructureName(String structureName)`
Field to be mapped *	not applicable	`setFieldName(String fieldName)`
Method access	direct access	`setGetMethodName(String name)` `setSetMethodName(String name)`

Indirection	use indirection	`useIndirection()`
		`dontUseIndirection()`
		`setUsesIndirection(boolean usesIndirection)`

Privately owned relationship	independent	`independentRelationship()`
		`privateOwnedRelationship()`
		`setIsPrivateOwned(Boolean isPrivateOwned)`

Read only	read / write	`readWrite()`
		`readOnly()`
		`setIsReadOnlylar(boolean readOnly)`

* Required property

8

Descriptor Implementation

A descriptor is a TopLink object that describes how an object's attributes and relationships are to be represented in relational database table(s). A "TopLink descriptor" is not the same as an "EJB deployment descriptor", although it plays a similar role.

Most descriptors can be created in the Mapping Workbench, but you may also have reasons to specify them in native Java code. This chapter illustrates

- Implementing primary keys in Java
- Implementing inheritance in Java
- Implementing interfaces in Java
- Setting the copy policy using Java
- Implementing multiple tables in Java code
- Implementing sequence numbers in Java
- Overriding the instantiation policy using Java code
- Implementing locking in Java
- Implementing identity maps in Java
- Implementing query keys in Java
- Implementing indirection in Java
- Implementing proxy indirection in Java
- Implementing object-relational descriptors in Java
- Changing Java classes to use indirection
- Setting the wrapper policy using Java code
- Implementing events using Java

Implementing primary keys in Java

If a single field constitutes the primary key, send the `setPrimaryKeyFieldName()` message to the descriptor. For a composite primary key, send the `addPrimaryKeyFieldName()` message for each field that makes up the primary key.

Alternatively, you could use a convenience method, `setPrimaryKeyFieldNames( )`, sending a `Vector` of the fields used as the primary key.

Example 8–1 Setting a single-field primary key in Java.

```
// Define a new descriptor and set the primary key.
descriptor.setPrimaryKeyFieldName("ADDRESS_ID");
```

Example 8–2 Setting a composite primary key in Java.

```
// Define a new descriptor and set the primary key.
descriptor1.addPrimaryKeyFieldName("PHONE_NUMBER");
descriptor1.addPrimaryKeyFieldName("AREA_CODE");
```

Implementing inheritance in Java

To implement an inheritance hierarchy completely in Java, you must modify the descriptors for the superclass and its subclasses. The inheritance implementation for a descriptor is encapsulated in an `InheritancePolicy` object, which is accessed by sending `getInheritancePolicy()` to the descriptor.

- Unless a class extraction method is used, send the `setClassIndicatorFieldName()` message to the `InheritancePolicy` of the root class. The parameter is a string indicating the table column that holds the subclass type information.
- In the root class, define the values that are written to the database which indicate the class type. You can do this by:
  - Sending the `addClassIndicator()` message for each of the instantiable subclasses in the hierarchy. This message requires two parameters — the indicator value and the subclass it represents.
  - Sending the `useClassNameAsIndicator()` message. This causes the full name of the class to be stored in the class indicator field.
- Send the `setParentClass()` message to the descriptor for each subclass.
- The root and branch classes can be configured so that queries return only instances of itself by calling the `dontReadSubclassesOnQueries()` method.
- Descriptors that inherit table names from a parent are not sent the `setTableName()` and `addTableName()` messages for the tables they inherit. Only the root class defines the primary key.

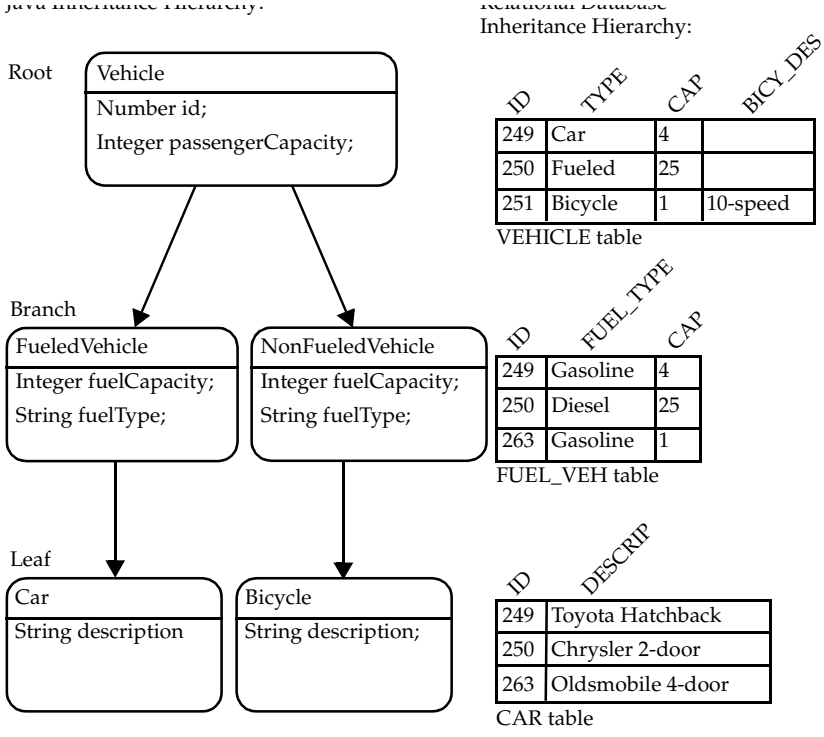
Queries for inherited superclasses can require multiple queries to obtain all of the rows for all of the subclasses. This is only required if the superclass is configured to read subclasses and its subclasses define additional tables. This situation can be optimized by providing TopLink with a view to execute the query against. This view can internally perform an outer join or union on all of the subclass tables and return a single result set with all of the data. This view can be set through the `setReadAllSubclassesViewName()` method.

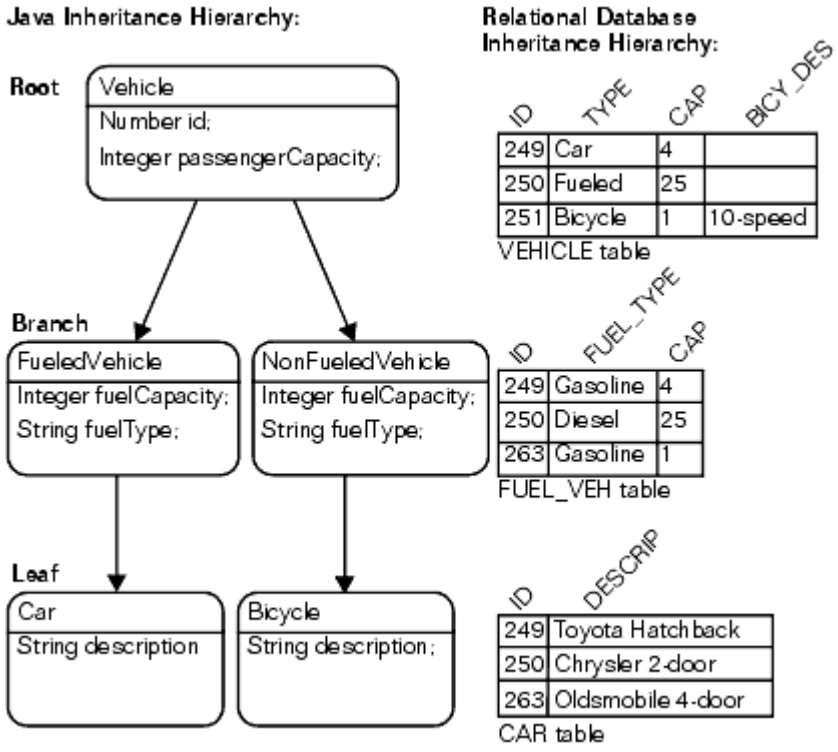
Using TopLink's default inheritance mechanism may not always be possible. In this case the inheritance mechanism can be further customized. Instead of using a class indicator field and mapping, a class extraction method may be used. This method takes the objects row and returns the class to be used for that row. The `setClassExtractionMethodName()` method is used to accomplish this.

Normally queries for inherited classes also require filtering of the tables rows; by default, TopLink generates this from the class indicator information. However, if the class extraction method is given, the filtering expressions must be specified. These can be set for concrete classes through `setOnlyInstancesExpression()` and for branch classes through `setWithAllSubclassesExpression()`.

Figure 8–1 shows an example of an inheritance hierarchy. The `Vehicle-Bicycle` branch demonstrates how all subclass information can be stored in one table. The `FueledVehicle-Car` branch demonstrates how subclass information can be stored in two tables.

Figure 8–1 Inheritance hierarchy





The `Car` and `Bicycle` classes are leaf classes, so queries done on them return instances of `Car` and `Bicycle` respectively.

`FueledVehicle` is a branch class. By default, branch classes are configured to read instances and subclass instances. Queries for `FueledVehicle` return instances of `FueledVehicle` and also return instances of `Car`.

`NonFueledVehicle` is a branch class and is configured to read subclasses. Because it does not have a class indicator defined in the root, it cannot be written to the database. Queries done on `NonFueledVehicle` return instances of its subclasses.

`Vehicle` is a root class, which is configured to read instances of itself and instances of its subclass by default. Queries made on the `Vehicle` class return instances of any of the concrete classes in the hierarchy.

Example 8–3 Implementing descriptors for the different classes in the inheritance hierarchy.

```
// Vehicle is a root class. Because it is the root class, it must add the class
indicators for its subclasses.
public static Descriptor descriptor()
{
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(Vehicle.class);
descriptor.setTableName("VEHICLE");
descriptor.setPrimaryKeyFieldName("ID");

// Class indicators must be supplied for each of the subclasses in the hierarchy
that can have instances.
InheritancePolicy policy = descriptor.getInheritancePolicy();
policy.setClassIndicatorFieldName("TYPE");
policy.addClassIndicator(FueledVehicle.class, "Fueled");
policy.addClassIndicator(Car.class, "Car");
policy.addClassIndicator(Bicycle.class, "Bicycle");

descriptor.addDirectMapping("id", "ID");
descriptor.addDirectMapping("passengerCapacity", "CAP");

return descriptor;
}

// FueledVehicle descriptor; it is a branch class and a subclass of Vehicle.
Queries made on this class will return instances of itself and instances of its
subclasses.
public static Descriptor descriptor()
{
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(FueledVehicle.class);
descriptor.addTableName("FUEL_VEH");
descriptor.getInheritancePolicy().setParentClass(Vehicle.class);
descriptor.addDirectMapping("fuelCapacity", "FUEL_CAP");
descriptor.addDirectMapping("fuelType", "FUEL_TYPE");
return descriptor;
}

// Car descriptor; it is a leaf class and subclass of FueledVehicle.
public static Descriptor descriptor()
{
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(Car.class);
descriptor.addTableName("CAR");
```

```
descriptor.getInheritancePolicy().setParentClass(FueledVehicle.class);

// Next define the attribute mappings.
descriptor.addDirectMapping("description", "DESCRIP");
descriptor.addDirectMapping("fuelType", "FUEL_VEH.FUEL_TYPE");
return descriptor;
}

// NonFueledVehicle descriptor; it is a branch class and a subclass of Vehicle.
Queries made on this class will return instances of its subclasses.
public static Descriptor descriptor()
{
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(NonFueledVehicle.class);
descriptor.getInheritancePolicy().setParentClass(Vehicle.class);
return descriptor;
}

// Bicycle descriptor; it is a leaf class and subclass of NonFueledVehicle.
public static Descriptor descriptor()
{
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(Bicycle.class);
descriptor.getInheritancePolicy().setParentClass(NonFueledVehicle.class);
descriptor.addDirectMapping("description", "BICY_DES");
return descriptor;
}

// FueledVehicle class; If a class extraction method is used, the following
would need to be added to specify that only the branch class itself needs to be
returned. This example is just specifying the class indicator field, which can
also be specified in Mapping Workbench in the Descriptor Advanced Properties
dialog.
public void addToDescriptor(Descriptor descriptor)
{
ExpressionBuilder builder = new ExpressionBuilder();
descriptor.getInheritancePolicy().setOnlyInstancesExpression(builder.getField("V
EHICLE.TYPE").equal("F"));
}

// FueledVehicle class; If a class extraction method is used, the following
would need to be added to specify that the branch class and its subclasses need
to be returned. This example can also be specified in Mapping Workbench in the
Descriptor Advanced Properties dialog.
public void addToDescriptor (Descriptor descriptor)
```

```
{
ExpressionBuilder builder = new ExpressionBuilder();
descriptor.getInheritancePolicy().withAllSubclassesExpression(builder.getField("
VEHICLE.TYPE").equal("F"));
}
```

Reference

Table 8–1 summarizes the most common public methods for `InheritancePolicy`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for `InheritancePolicy`, see the TopLink JavaDocs.

Table 8–1 Elements for the inheritance policy

Element	Default	Method Names
Class indicators	use indicator mapping	`setClassIndicatorFieldName` **`(String fieldName)`**
Parent classes	not applicable	`setParentClass` **`(Class parentClass)`**

Implementing interfaces in Java

Descriptors can their own parent interfaces. They can set multiple interfaces if they have implemented multiple interfaces. The query keys are defined in a normal way except that they must define the abstract query key from the interface descriptor in their descriptors. An abstract query key on the interface descriptor enables it to write expression queries on the interface.

Example 8–4 Using an abstract query key on the interface descriptor.

```
ExpressionBuilder contact = new
ExpressionBuilder();session.readObject(Contact.class,
contact.get("id").equal(2));
```

Setting the copy policy using Java

The `Descriptor` class provides three methods that can be used to determine how an object is cloned:

- `useInstantiationCopyPolicy()` — the default method; TopLink creates a new instance of the object using the technique indicated by the descriptor's instantiation policy. The default behavior is to use the default constructor. The new instance is then populated by using the descriptor's mappings to copy attributes from the original object.

  Note: `Descriptor.useInstantiationCopyPolicy()` replaces `Descriptor.useConstructorCopyPolicy()` available in previous versions of TopLink. The old method is still supported, but it has been deprecated.

- `useCloneCopyPolicy()` — TopLink calls the `clone()` method of the object; you must ensure that the clone method is written correctly and returns a logical shallow clone of the object
- `useCloneCopyPolicy(String)` — this method is called by passing in a string that contains the name of a method that clones the object; you must ensure that the method specified returns a logical shallow clone of the object

Implementing multiple tables in Java code

To define a multiple table descriptor, call the `addTableName()` method for each table the descriptor maps to. If the descriptor inherits its primary table and is only defining a single additional one, then the descriptor is mapped normally to this table.

Primary keys match

Normally the primary key is defined only for the primary table of the descriptor. The primary table is the first table specified through `addTableName()`. The primary key is not defined for the additional tables and is required to be the same as in the primary table. If the additional table's key is different, refer to the next example.

By default, all the fields in a mapping are assumed to be part of the primary table. If a mapping's field is for one of the additional tables it must be fully qualified with the field's table name.

Example 8–5 Implementing a multiple table descriptor where the primary keys match.

```
//Define a new descriptor that uses three tables.
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(Employee.class);
descriptor.addTableName("PERSONNEL"); // Primary table
descriptor.addTableName("EMPLOYMENT");
descriptor.addTableName("USERS");

descriptor.addPrimaryKeyFieldName("PER_NUMBER");
descriptor.addPrimaryKeyFieldName("DEP_NUMBER");

descriptor.addDirectMapping("id", "PER_NUMBER");
descriptor.addDirectMapping("firstName", "F_NAME");
descriptor.addDirectMapping("lastName", "L_NAME");

OneToOneMapping department = new OneToOneMapping();
department.setAttributeName("department");
department.setReferenceClass(Department.class);
department.setForeignKeyFieldName("DEP_NUMBER");
descriptor.addMapping(department);
// Mapping the primary key fields in the additional tables is not required
descriptor.addDirectMapping("salary", "EMPLOYMENT.SALARY");

AggregateObjectMapping period = new AggregateObjectMapping();
period.setAttributeName(period);
period.setReferenceClass(EmployementPeriod.class);
period.addFieldNameTranslation("EMPLOYMENT.S_DATE", "S_DATE");
period.addFieldNameTranslation("EMPLOYMENT.E_DATE", "E_DATE");
descriptor.addMapping(period);

descriptor.addDirectMapping("userName", "USERS.NAME");
descriptor.addDirectMapping("password", "USERS.PASSWORD");
```

Primary keys are named differently

If the additional tables primary key is named differently then call the descriptor method `addMultipleTablePrimaryKeyName()`, which provides:

- The field of the primary key from the primary table
- The additional table name
- The field in the additional table that the primary key maps to

Example 8–6 Implementing a multiple table descriptor where the additional table primary keys are named differently.

```
//Define a new descriptor that uses three tables.
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(Employee.class);
descriptor.addTableName("PERSONNEL");
// Primary table
descriptor.addTableName("EMPLOYMENT");
descriptor.addTableName("USERS");

descriptor.addPrimaryKeyFieldName("PER_NUMBER");
descriptor.addPrimaryKeyFieldName("DEP_NUMBER");

descriptor.addMultipleTablePrimaryKeyName("PERSONEL.PER_NUMBER",
"USERS.PERSONEL_NO");
descriptor.addMultipleTablePrimaryKeyName("PERSONEL.DEP_NUMBER",
"USERS.DEPARTMENT_NO");

// Assumed EMPLOYMENT uses same primary key
descriptor.addDirectMapping(id, PER_NUMBER);

OneToOneMapping department = new OneToOneMapping();
department.setAttributeName("department");
department.setReferenceClass(Department.class);
department.setForeignKeyFieldName("DEP_NUMBER");
descriptor.addMapping(department);

// Primary key does not have to be mapped for additional tables.
...
```

Tables related by foreign key relationships

For TopLink to support read, insert, update and delete operations on an object mapped to multiple tables:

- The foreign key information must be explicitly specified on the descriptor
- The foreign keys and primary keys must be mapped in the object

The API is `addMultipleTableForeignKeyFieldName()`. This method builds the join expression and adjusts the table insertion order to respect the foreign key constraints.

The following example shows the setup of a descriptor for an object mapped to multiple tables where the tables are related by a foreign key relationship from the

primary table to the secondary table. The `addMultipleTableForeignKeyFieldName()` method is used to specify the direction of the foreign key relationship.

If the foreign key is in the secondary table and refers to the primary table then the order of the arguments to `addMultipleTableForeignKeyFieldName()` is reversed.

> **Note:** The foreign key field in the primary table and the primary key in the secondary table must be mapped. This allows read, insert, update and delete operation to be performed on the Employee object.

Example 8–7 Implementing multiple tables where a foreign key from the primary table to the secondary table is used to join the tables.

```
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(Employee.class);
Vector vector = new Vector();
vector.addElement("EMPLOYEE");
vector.addElement(ADDRESS");
descriptor.setTableNames(vector);
descriptor.addPrimaryKeyFieldName("EMPLOYEE.EMP_ID");
// Map the foreign key field of the employee table and the primary key of the
address table.
descriptor.addDirectMapping("employee_addressID", "EMPLOYEE.ADDR_ID");
descriptor.addDirectMapping("address_addressID", "ADDRESS.ADDR_ID");

// Setup the join from the address table to the country employee table to the
address table by specifying the FK info to the descriptor. Set the foreign key
info from the address table to the country table.
descriptor.addMultipleTableForeignKeyFieldName("EMPLOYEE.ADDR_ID",
"ADDRESS.ADDR_ID");
```

Non-standard table relationships

Occasionally the join condition can be non-standard. In this case, the descriptors query manager can be used to provide a custom multiple table join expression. The `getQueryManager()` method is called on the descriptor to obtain its query manager, and the `setMultipleTableJoinExpression()` method is used to customize the join expression.

Simply specifying the join expression allows TopLink to perform read operations for the object. Insert operations can also be supported if the table insertion order is specified and the primary key of the additional tables is mapped manually.

The insertion order is required so as not to violate foreign key constraints when inserting to the multiple tables. The insert order can be specified using the descriptor method `setMultipleTableInsertOrder()`.

The following example shows the use of the `setMultipleTableJoinExpression()` and `setMultipleTableInsertOrder()` methods. In addition, it shows the use of a custom join expression without specifying the table insert order.

> **Note:** Using these methods does not support update or delete operations because of the lack of primary key information for the secondary table(s). If update and delete operations are required they could be done with custom SQL, or the foreign key information can be specified explicitly as explained in the previous section.

Example 8–8 Implementing multiple tables where a join expression and the table insert order are specified.

Using this method allows only read and insert operations to be performed on Employee objects. Note that the primary key of the secondary table and the foreign key of the primary table must be mapped and maintained by the application for insert operations to work.

```
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(Employee.class);
Vector vector = new Vector();
vector.addElement("EMPLOYEE");
vector.addElement(ADDRESS");
descriptor.setTableNames(vector);

// Specify the primary key information for each table.
descriptor.addPrimaryKeyFieldName("EMPLOYEE.EMP_ID");

// Map the foreign key field of the employee table and the primary key of the
address table.
descriptor.addDirectMapping("employee_addressID", "EMPLOYEE.ADDR_ID");
descriptor.addDirectMapping("address_addressID", "ADDRESS.ADDR_ID");
// Setup the join from the employee table to the address table using a custom
join expression and specifying the table insert order.
ExpressionBuilder builder = new ExpressionBuilder();
```

```
descriptor.getQueryManager().setMultipleTableJoinExpression(builder.getField("EM
PLOYEE.ADDR_ID").equal(builder.getField("ADDRESS.ADDR_ID")));
Vector tables = new Vector(2);
tables.addElement(new DatabaseTable("ADDRESS"));
tables.addElement(new DatabaseTable("EMPLOYEE"));
descriptor.setMultipleTableInsertOrder(tables);
...
```

Example 8–9 Mapping a multiple table descriptor where a custom join expression is required.

In this example only read operations are supported.

```
//Define a new descriptor that uses three tables.
Descriptor descriptor = new Descriptor();
descriptor.setJavaClass(Employee.class);
descriptor.addTableName("PERSONNEL");
// Primary table
descriptor.addTableName("EMPLOYMENT");
descriptor.addPrimaryKeyFieldName("PER_NO");
descriptor.addPrimaryKeyFieldName("DEP_NO");

ExpressionBuilder builder = new ExpressionBuilder();
descriptor.getQueryManager().setMultipleTableJoinExpression((builder.getField("P
ERSONEL.EMP_NO").equal(builder.getField("EMPLOYMENT.EMP_NO")));
descriptor.addDirectMapping("personelNumber", "PER_NO");

OneToOneMapping department = new OneToOneMapping();
department.setAttributeName("department");
department.setReferenceClass(Department.class);
department.setForeignKeyFieldName("DEP_NO");
descriptor.addMapping(department);
// The primary key field on the EMPLOYMENT does not have to be mapped.
...
```

Implementing sequence numbers in Java

To implement sequence numbers using Java code, send the `setSequenceNumberFieldName( )` message to the descriptor to register the name of the database field that holds the sequence number. The `setSequenceNumberName( )` method also holds the name of the sequence. This name can be one of the entries in the `SEQ_NAME` column or the name of the sequence object (if you are using Oracle native sequencing).

Notes:

- The sequence field must be in the first (primary) table if multiple tables are used.
- If you use Sybase, SQL Server or Informix native sequencing, this has no direct meaning but should still be set for compatibility reasons.

Example 8–10 Implementing sequence numbers using Java code.

```
// Set the sequence number information.
descriptor.setSequenceNumberName("EMP_SEQ");
descriptor.setSequenceNumberFieldName("EMP_ID");
```

Overriding the instantiation policy using Java code

The `Descriptor` class provides the following methods to specify how objects get instantiated.

- `useDefaultConstructorInstantiationPolicy()` instructs TopLink to use the default constructor to create new instances of objects built from the database. This method can be private, protected, or default/package.
- `useFactoryInstantiationPolicy(Object, String)` instructs TopLink to send the message specified by the `String` parameter to an object factory specified by the `Object` parameter to create objects from the database. The object factory method that is used can be public, private, protected, or default/package, and requires no arguments.
- `useMethodInstantiationPolicy(String)` instructs TopLink to send the message contained in the string parameter to create objects that are populated with data from the database. This method can be a public, static method on the descriptor class, or it can be private, protected, or default/package. It must return a new instance of the class.
- `useFactoryInstantiationPolicy(Class factoryClass, String methodName)` instructs TopLink to send the message contained in the `String` parameter to an instance of the specified `factoryClass`. This method must be return a new instance of the descriptor class. TopLink instantiates the factory by invoking the default constructor of the specified `factoryClass`. Both the `factoryClass` default constructor and the method invoked on the factory can be private, protected, or default/package.]

- `useFactoryInstantiationPolicy(Class factoryClass, String methodName, String factoryMethodName)` instructs TopLink to send the message contained in the first `String` parameter, `methodName`, to an instance of the specified `factoryClass`. This method must return a new instance of the descriptor class. TopLink instantiates the factory by invoking the second `String`, `methodName`, on the specified `factoryClass`. This method must be a static method on the `factoryClass` and must return an instance of the `factoryClass`. The factory class static factory method and the method invoked on the factory can be private, protected, or default/package.

Implementing locking in Java

Use the API to set optimistic locking completely in code. All of the API is on the descriptor:

- `useVersionLocking(String)` sets this descriptor to use version locking, and increments the value in the specified field name for update or delete
- `useChangedFieldsLocking()` tells this descriptor to compare only modified fields for an update or delete
- `useTimestampLocking(String)` sets this descriptor to use timestamp locking and writes the current server time in the field every update or delete
- `useAllFieldsLocking()` tells this descriptor to compare every field for an update or delete
- `useSelectedFieldsLocking(Vector)` tells this descriptor to compare the field names specified in this vector of Strings for an update or delete

Example 8–11 Implementing optimistic locking using the VERSION field of EMPLOYEE table as the version number of the optimistic lock.

```
// Set the field that control optimistic locking. No mappings are set for fields
which are version fields for optimistic locking.
descriptor.useVersionLocking("VERSION");
```

The code in the example above, stores the optimistic locking value in the identity map. If the value should be stored in a non-read only mapping, then the code would be:

```
descriptor.useVersionLocking("VERSION", false);
```

The `false` indicates that the lock value is not stored in the cache but is stored in the object.

Implementing identity maps in Java

To change the identity map type for a descriptor from the default to specify it explicitly as full identity map), `useNoIdentityMap()`, `useCacheIdentityMap()`, `useWeakIdentityMap()`, `useSoftCacheWeakIdentityMap()`, `useHardCacheWeakIdentityMap()` or `useFullIdentityMap()` message to the descriptor.

To change the size of the identity map from the default of **100**, use the `setIdentityMapSize()` message.

Example 8–12 Changing the default identity map parameters

```
// Set any identity map parameters.
descriptor.useCacheIdentityMap();
descriptor.setIdentityMapSize(10);
```

Implementing query keys in Java

Register query keys with a descriptor using the `addQueryKey()` method of the `Descriptor` class. Direct query keys can also be defined with the `addDirectQueryKey()` method, specifying the name of the query key and the name of the table field. Abstract query keys are registered to interface descriptors through the `addAbstractQueryKey()` method.

Example 8–13 Implementing a query key.

```
// Add a query key for the foreign key field using the direct method
direct descriptor.addDirectQueryKey("managerId", "MANAGER_ID");

// The same query key could also be added through the add method
DirectQueryKey directQueryKey = new DirectQueryKey();
directQueryKey.setName("managerId");
directQueryKey.setFieldName("MANAGER_ID");
descriptor.addQueryKey(directQueryKey);

// Add a one-to-one query key for the large project that the employee is a
leader of (this assumes only one project)
OneToOneQueryKey lprojectQueryKey = new OneToOneQueryKey();
lprojectQueryKey.setName("managedLargeProject");
lprojectQueryKey.setReferenceClass(LargeProject.class);
ExpressionBuilder builder = new ExpressionBuilder();
projectQueryKey.setJoinCriteria(builder.getField("PROJECT.LEADER_
ID").equal(builder.getParameter("EMPLOYEE.EMP_ID")));
```

```
descriptor.addQueryKey(lprojectQueryKey);

// Add a one-to-many query key for the projects that the employee multiple
projects)
OneToManyQueryKey projectsQueryKey = new OneToManyQueryKey();
projectsQueryKey.setName("managedProjects");
projectsQueryKey.setReferenceClass(Project.class);
ExpressionBuilder builder = new ExpressionBuilder();
projectsQueryKey.setJoinCriteria(builder.getField("PROJECT.LEADER_
ID").equal(builder.getParameter("EMPLOYEE.EMP_ID")));
descriptor.addQueryKey(projectsQueryKey);
// Next define the mappings.
...
```

Implementing indirection in Java

To create indirection objects in code, the application must replace the relationship reference with a `ValueHolderInterface`. It must also call the `useIndirection()` method of the mapping if the mapping does not use indirection by default. Likewise, call the `dontUseIndirection()` method to disable indirection. `ValueHolderInterface` is defined in the `oracle.toplink.indirection`.

Example 8–14 A mapping that does not use indirection.

```
// Define the One-to-One mapping. Note that One-to-One mappings have indirection
enabled by default, so the "dontUseIndirection()" method must be called if
indirection is not used.
OneToOneMapping oneToOneMapping = new OneToOneMapping();
oneToOneMapping.setAttributeName("address");
oneToOneMapping.setReferenceClass(Address.class);
oneToOneMapping.setForeignKeyFieldName("ADDRESS_ID");
oneToOneMapping.dontUseIndirection();
oneToOneMapping.setSetMethodName("setAddress");
oneToOneMapping.setGetMethodName("getAddress");
descriptor.addMapping(oneToOneMapping);
```

The following code illustrates a mapping using indirection.

```
// Define the One-to-One mapping. One-to-One mappings have indirection enabled
by default, so the "useIndirection()" method is unnecessary if indirection is
used.
OneToOneMapping oneToOneMapping = new OneToOneMapping();
oneToOneMapping.setAttributeName("address");
oneToOneMapping.setReferenceClass(Address.class);
```

```
oneToOneMapping.setForeignKeyFieldName("ADDRESS_ID");
oneToOneMapping.setSetMethodName("setAddressHolder");
oneToOneMapping.setGetMethodName("getAddressHolder");
descriptor.addMapping(oneToOneMapping);
```

Implementing proxy indirection in Java

To enable proxy indirection in Java code, use the following API for `ObjectReferenceMapping`:

- `useProxyIndirection()` – indicates that TopLink should use proxy indirection for this mapping. When the source object is read from the database, a proxy for the target object is created and used in place of the "real" target object. When any method other than to`String()` is called on the proxy, the "real" data will be read from the database.

Example 8–15 Using proxy indirection.

```
// Define the 1:1 mapping, and specify that Proxy Indirection should be used
OneToOneMapping addressMapping = new OneToOneMapping();
addressMapping.setAttributeName("address");
addressMapping.setReferenceClass(AddressImpl.class);
addressMapping.setForeignKeyFieldName("ADDRESS_ID");
addressMapping.setSetMethodName("setAddress");
addressMapping.setGetMethodName("getAddress");
addressMapping.useProxyIndirection();
descriptor.addMapping(addressMapping);
. . .
```

Implementing object-relational descriptors in Java

Use the `ObjectRelationalDescriptor` class to define object-relational descriptors. This descriptor subclass contains the following additional properties:

- **Structure name:** Name of the object-type structure representing the class
- **Field ordering:** Field index of the object-type (required because object-type can be returned through JDBC as indexed arrays)

The demo application provided in `<INSTALL_DIR>\examples\core\examples\sessions\remote\rmi` illustrates an object-relational data model and descriptors.

Example 8–16 Creating an object-relational descriptor.

```
import oracle.toplink.objectrelational.*;
ObjectRelationalDescriptor descriptor = new ObjectRelationalDescriptor()
descriptor.setJavaClass(Employee.class);
descriptor.setTableName("EMPLOYEES");
descriptor.setStructureName("EMPLOYEE_T");
descriptor.setPrimaryKeyFieldName("OBJECT_ID");

descriptor.addFieldOrdering("OBJECT_ID");
descriptor.addFieldOrdering("F_NAME");
descriptor.addFieldOrdering("L_NAME");
descriptor.addFieldOrdering("ADDRESS");
descriptor.addFieldOrdering("MANAGER");
descriptor.addDirectMapping("id", "OBJECT_ID");
descriptor.addDirectMapping("firstName", "F_NAME");
descriptor.addDirectMapping("lastName", "L_NAME");
//Refer to the mappings section for examples of object relational mappings.
...
```

Changing Java classes to use indirection

Attributes using indirection must conform to the `ValueHolderInterface`. You can change your attribute types in the Class Editor without re-importing your Java classes. Ensure that you change the attribute types in your Java code as well. Attributes that are typed incorrectly will be marked as deficient.

In addition to changing the attribute's type, you may also need to change its accessor methods. If you use method access, TopLink requires accessors to the *indirection* object itself, so your get method returns an instance that conforms to `ValueHolderInterface` and your set method accepts one argument that conforms to the same. If the instance variable returns a Vector instead of an object then the value holder should be defined in the constructor as follows:

```
addresses = new ValueHolder(new Vector());
```

In any case, the application uses the `getAddress()` and `setAddress()` methods to access the `Address` object. When indirection is used, TopLink uses the `getAddressHolder()` and `setAddressHolder()` methods when saving and retrieving instances to and from the database.

Example 8–17 Implementing the Employee class using indirection with method access for a one-to-one mapping to Address.

The class definition is modified so that the address attribute of `Employee` is a `ValueHolderInterface` instead of an `Address` and appropriate get and set methods are supplied.

```
// Initialize ValueHolders in Employee Constructor
public Employee() {
address = new ValueHolder();
}
protected ValueHolderInterface address;

// 'Get' and 'Set' accessor methods registered with the mapping and used by
TopLink.
public ValueHolderInterface getAddressHolder() {
return address;
}
public void setAddressHolder(ValueHolderInterface holder) {
address = holder;
}

// 'Get' and 'Set' accessor methods used by the application to access the
attribute.
public Address getAddress() {
return (Address) address.getValue();
}
public void setAddress(Address theAddress) {
address.setValue(theAddress);
}
```

Setting the wrapper policy using Java code

The `Descriptor` class provides methods that are used in conjunction with the wrapper policy:

- `setWrapperPolicy(oracle.toplink.descriptors.WrapperPolicy)` can be invoked to provide a wrapper policy for the descriptor
- `getWrapperPolicy()` returns the wrapper policy for a descriptor

Implementing events using Java

In this example, we check to see if there is a lock conflict whenever an instance of `Employee` is built from information in the database:

```
//In the employee class, declare the event method which will be invoked when the
event occurs.
public void postBuild(DescriptorEvent event) {
    // Uses objects row to integrate with some application level locking
    service.
    if ((event.getRow().get("LOCKED")).equals("T")) {
        LockManager.checkLockConflict(this);
    }
}
```

Registering event listeners

TopLink only supports registering events to call methods on the domain object. If you want an object other than the domain object to handle these events, you must register it as a *listener* with the descriptor's event manager. If you want a LockManager to receive events for all `Employees`, you could modify your descriptor amendment to register the LockManager as the listener. Any object you register as a listener must implement the `DescriptorEventListener` interface. The amendment method is shown in the following example.

Example 8–18 Registering event listeners

```
public static void addToDescriptor(Descriptor descriptor)
{
    descriptor.getEventManager().addListener
    (LockManager.activeManager());
}
```

Reference

Table 8–2 summarizes the most common public methods for `DescriptorEventManager`:

- the Default column describes default settings of the descriptor element
- in the Method Names column, arguments are bold, methods are not

For a complete description of all available methods for `DescriptorEventManager`, see the TopLink JavaDocs.

Table 8–2 Elements for the descriptor event manager

Element	Default	Method Names
Events selectors Defaults come from listener interface implementation	All events take `DescriptorEvent`	All events take **`(String methodName)`**
	`postBuild`	`setPostBuildSelector`
	`postRefresh`	`setPostRefreshSelector`
	`preWrite`	`setPreWriteSelector`
	`postWrite`	`setPostWriteSelector`
	`preDelete`	`setPreDeleteSelector`
	`postDelete`	`setPostDeleteSelector`
	`preInsert`	`setPreInsertSelector`
	`postInsert`	`setPostInsertSelector`
	`preUpdate`	`setPreUpdateSelector`
	`postUpdate`	`setPostUpdateSelector`
	`aboutToInsert`	`setAboutToInsertSelector`
	`aboutToUpdate`	`setAboutToUpdateSelector`
	`postClone`	`setPostCloneSelector`
	`postMerge`	`setPostMergeSelector`
Listener registration	Source object if it implements the listener interface	`addListener` **`(DescriptorEventListener listener)`**

Descriptor-Event reference (available methods on Descriptor-Event)		`getSource()`
		`getSession()`
		`getQuery()`
		`getDescriptor()`

	only; `aboutToInsert/` Update, / Build	`getRow()`
	only; postMerge / Clone / write events within a unit of work	`getOriginalObject()`

A

Sessions.xml DTD

A document type definition or DTD is file that defines how the markup tags in an XML documents should be interpreted. Following is the DTD for TopLink's sessions.xml file.

sessions.xml dtd

<?xml version='1.0' encoding='UTF-8' ?>

<!--This is the root element and exists only for XML structure-->

<!ELEMENT toplink-configuration (session* , session-broker*)>

<!--This element used if a session broker must be configured-->

<!ELEMENT session-broker (name , session-name+)>

<!--This is the element that represents the session name-->

<!ELEMENT session-name (#PCDATA)>

<!--This is the node element that describes a particular session for use within toplink-->

<!ELEMENT session (name , (project-class | project-xml) , session-type , login? , cache-synchronization-manager? , event-listener-class*, profiler-class?, external-transaction-controller-class?, exception-handler-class?, connection-pool*, enable-logging?, logging-options?)>

<!--This is the type of session that is being configured-->

<!ELEMENT session-type (server-session | database-session)>

<!ELEMENT server-session EMPTY>

<!ELEMENT database-session EMPTY>

<!--This is the class name that this session will load to provide login and mapping information-->

<!ELEMENT project-class (#PCDATA)>

<!--This is the file that contains the project that this session will load to provide login and mapping information-->

<!ELEMENT project-xml (#PCDATA)>

<!--This is the element that is used if the session will be synchronized with others-->

<!ELEMENT cache-synchronization-manager (clustering-service , multicast-port? , multicast-group-address? , packet-time-to-live? , is-asynchronous? , should-remove-connection-on-error? , (jndi-user-name , jndi-password)? , naming-service-url)>

<!--This is the name of the clustering service that will be used for connecting sessions for Cache Synchronization-->

<!ELEMENT clustering-service (#PCDATA)>

<!--This is the IP that the Clustering Service will be listening for new session anouncements-->

<!ELEMENT multicast-group-address (#PCDATA)>

<!--This is the multicast port the the clustering service will be listening on for announcements of new sessions-->

<!ELEMENT multicast-port (#PCDATA)>

<!ATTLIST multicast-port e-dtype NMTOKEN #FIXED 'number'>

<!--Set to true if synchronization should not wait until all sessions have been synchronised before returning-->

<!ELEMENT is-asynchronous (#PCDATA)>

<!--Set to true if the connection should be removed from this session if a communication error occurs-->

<!ELEMENT should-remove-connection-on-error (#PCDATA)>

<!--The URL to the global Namespace for the Synchronization connection. Usually the URL of the JNDI service-->

<!ELEMENT naming-service-url (#PCDATA)>

<!--The maximum number of hops a packet will be broadcast-->

<!ELEMENT packet-time-to-live (#PCDATA)>

<!ATTLIST packet-time-to-live e-dtype NMTOKEN #FIXED 'number' >

<!--This element used if a user name is required to access the JNDI service in the case of Cache Synchronization-->

<!ELEMENT jndi-user-name (#PCDATA)>

<!--This element used if a password is required to access the JNDI service in the case of Cache Synchronization-->

<!ELEMENT jndi-password (#PCDATA)>

<!--This describes one of possibly many event-listeners that can be registered on the session-->

<!ELEMENT event-listener-class (#PCDATA)>

<!--This element represents the class name of the profiler that will be used by the session-->

<!ELEMENT profiler-class (#PCDATA)>

<!--This is the class that the session will use as the external transaction controller-->

<!ELEMENT external-transaction-controller-class (#PCDATA)>

<!--This is the class that the session will use to handle exceptions generated from within the session-->

<!ELEMENT exception-handler-class (#PCDATA)>

<!--SQL will be logged to the Session writer which, by default, is System.out-->

<!ELEMENT enable-logging (#PCDATA)>

<!--This element used to specify the extra logging options-->

<!ELEMENT logging-options (log-debug? , log-exceptions? , log-exception-stacktrace? , print-thread? , print-session? , print-connection? , print-date?)>

<!--Debug messages will be logged-->

<!ELEMENT log-debug (#PCDATA)>

<!--exceptions will be logged-->

<!ELEMENT log-exceptions (#PCDATA)>

<!--exceptions stack traces will be logged when they occur-->

<!ELEMENT log-exception-stacktrace (#PCDATA)>

<!--Each line of the log will contain the connection id-->

<!ELEMENT print-connection (#PCDATA)>

<!--each line of the log will contain the date-->

<!ELEMENT print-date (#PCDATA)>

<!--each line of the log will contain the session id-->

<!ELEMENT print-session (#PCDATA)>

<!--each line of the log will contain the thread id-->

<!ELEMENT print-thread (#PCDATA)>

<!--This the node element that stores the information for the connection pools-->

<!ELEMENT connection-pool (is-read-connection-pool , name , max-connections? , min-connections? , login)>

<!ELEMENT is-read-connection-pool (#PCDATA)>

<!--The max number of connections that will be created in the pool-->

<!ELEMENT max-connections (#PCDATA)>

<!ATTLIST max-connections e-dtype NMTOKEN #FIXED 'number' >

<!--The min number of connections that will aways be in the pool-->

<!ELEMENT min-connections (#PCDATA)>

<!ATTLIST min-connections e-dtype NMTOKEN #FIXED 'number' >

<!--This is the node element that represents the login for a particular connection pool. The read and write connection pools will use the login from the project-->

<!ELEMENT login (license-path? , driver-class? , (connection-url | datasource)? , (non-jts-connection-url | non-jts-datasource)? , platform-class? , user-name? , password? , uses-native-sequencing? , sequence-preallocation-size? , sequence-table? , sequence-name-field? , sequence-counter-field? , (should-bind-all-parameters , should-cache-all-statements?)? , uses-byte-array-binding? , uses-string-binding? , uses-streams-for-binding? , should-force-field-names-to-uppercase? , should-optimize-data-conversion? , should-trim-strings? , uses-batch-writing? , uses-jdbc20-batch-writing? ,

uses-external-connection-pool? , uses-native-sql? ,
uses-external-transaction-controller?)>

<!--Obsolete. The contents of this element are ignored at runtime. -->

<!ELEMENT license-path (#PCDATA)>

<!--This is the element that represents the platform class name-->

<!ELEMENT platform-class (#PCDATA)>

<!--This is the element that represents the database driver class name-->

<!ELEMENT driver-class (#PCDATA)>

<!--This is the URL that will be used to connect to the database-->

<!ELEMENT connection-url (#PCDATA)>

<!--This is the URL of a datasource that may be used by the session to connect to the database-->

<!ELEMENT datasource (#PCDATA)>

<!--This element is used in the login as well as the Cache Synchronization feature-->

<!ELEMENT user-name (#PCDATA)>

<!--This element is used in the login as well as the Cache Synchronization feature-->

<!ELEMENT password (#PCDATA)>

<!--Set to true if the login should use native sequencing-->

<!ELEMENT uses-native-sequencing (#PCDATA)>

<!--Sets the sequencing pre-allocation size. This is the number of sequences that will be retrieved from the database each time-->

<!ELEMENT sequence-preallocation-size (#PCDATA)>

<!ATTLIST sequence-preallocation-size e-dtype NMTOKEN #FIXED 'number' >

<!--The name of the sequence table-->

<!ELEMENT sequence-table (#PCDATA)>

<!--The field within the sequence table the stores that the sequence name-->

<!ELEMENT sequence-name-field (#PCDATA)>

<!--The field within the sequence table that stores the -->

<!ELEMENT sequence-count-field (#PCDATA)>

<!--Set to true if all queries should bind all parameters-->

<!ELEMENT should-bind-all-parameters (#PCDATA)>

<!--Set to true if all statements should be cached-->

<!ELEMENT should-cache-all-statements (#PCDATA)>

<!--Set to true if byte arrays should be bound-->

<!ELEMENT uses-byte-array-binding (#PCDATA)>

<!--Set to true if strings should be bound-->

<!ELEMENT uses-string-binding (#PCDATA)>

<!--Set to true if streams should be used when binding attributes-->

<!ELEMENT uses-streams-for-binding (#PCDATA)>

<!--Set to true if field names should be converted to uppercase when generating SQL-->

<!ELEMENT should-force-field-names-to-uppercase (#PCDATA)>

<!--Set to true if the session should optimize data conversions-->

<!ELEMENT should-optimize-data-conversion (#PCDATA)>

<!--Set to true if the connection should use native SQL-->

<!ELEMENT uses-native-sql (#PCDATA)>

<!--Set to true if trailing white spaces should be removed from strings-->

<!ELEMENT should-trim-strings (#PCDATA)>

<!--Set to true if the connection should batch the statements-->

<!ELEMENT uses-batch-writing (#PCDATA)>

<!--Set to true if the connection should use jdbc2.0 batch writing-->

<!ELEMENT uses-jdbc20-batch-writing (#PCDATA)>

<!--Set to true if the connection should use an external connection pool-->

<!ELEMENT uses-external-connection-pool (#PCDATA)>

<!--Set to true if the session will be using an external transaction controller-->

<!ELEMENT uses-external-transaction-controller (#PCDATA)>

<!--Genereic element used to describe a string that represents the name of an item-->

<!ELEMENT name (#PCDATA)>

<!--This element used if a non-jts connection is required (usually only required in an Application server when CacheSync is used-->

<!ELEMENT non-jts-connection-url (#PCDATA)>

<!--This element used if a non-jts connection is required (usually only required in an Application server when CacheSync is used-->

<!ELEMENT non-jts-datasource (#PCDATA)>

B

TopLink Development Tools

TopLink provides development tools that make the development, testing and debugging of TopLink applications easier. This section introduces these tools and discusses

- The Schema Manager
- Session management services
- The stored procedure generator
- The Session Console
- The Performance Profiler

The Schema Manager

The schema manager creates and modifies tables in a database from a Java application. The schema manager can also create sequence numbers on an existing database and generate stored procedures.

The session console inspects descriptors of a project file. The session console can connect to the database through TopLink, run custom SQL clauses, and view a log of query performance. It is a good practice to test your TopLink Mapping Workbench project with the session console before doing further development.

The Profiler generates a log of query performance against a TopLink session. This helps to pinpoint performance bottlenecks and makes debugging easier.

Using the Schema Manager to create tables

TopLink provides a high-level mechanism for the creation of database tables. This mechanism is database-independent and uses Java types rather than database types.

TopLink provides the `TableDefinition` class for the creation of new database table schema in an independent format. TopLink can then take this generic table definition and create the appropriate table and fields on the database. TopLink determines at runtime which database is being used and creates the appropriate fields for that database.

The schema manager's purpose is to allow for the database to be easily recreated and modified during development and testing. It also makes it possible for the schema to be ported, with 100% accuracy, to any other database.

TopLink's table creation mechanism does not take into account any optimization features provided on the given database and therefore should be used for prototyping purposes only.

Creating a table definition

The `TableDefinition` class encapsulates all the information necessary to create a new table, including the names and properties of a table and all of its fields.

`TableDefinition` has the following methods:

```
setName()
addField()
addPrimaryKeyField()
addIdentityField()
addForeignKeyConstraint()
```

All table definitions must call the `setName()` method to set the name of the table that is being described by the `TableDefinition`.

Adding fields to a table definition

Fields are added to the `TableDefinition` using the `addField()` method. The primary key of the table is added to the table definition using the `addPrimaryKeyField()` method instead of the `addField()` method. These methods have the definitions shown in the following example.

Example B–1 Adding fields to a table definition

```
addField(String fieldName, Class type);
addField(String fieldName, Class type, int fieldSize)
addField(String fieldName, Class type, int fieldSize,
    intfieldSubsize);
addPrimaryKeyField(String fieldName, Class type);
addPrimaryKeyField(String fieldName, Class type,
    int fieldSize);
addForeignKeyConstraint(String name, String sourceField,
    String targetField, String targetTable)
```

When declaring a field, the first parameter, `fieldName`, is the name of the field to be added to the table. The second parameter, `type`, is the type of field to be added. To maintain compatibility among different databases, a Java class is specified for the second parameter instead of a database field type. TopLink translates the Java class to the appropriate database field type based on the database upon which it is created. For example, the `String` class translates to the CHAR type when connecting to dBase. However, if connecting to Sybase the `String` class translates to VARCHAR.

If the field must specify a size, then the second method would be used and the size would be specified as the `fieldSize` parameter. If the field specifies a sub-size, the third method would be used to specify the `fieldSubSize` parameter.

Some databases require that a sub-size be specified for a given field type whereas another database vendor does not. TopLink automatically checks for this and removes or adds the sub-size if necessary. This allows the table creation mechanism to remain generic across multiple database vendors.

Defining Sybase and SQL Server sequence numbers

If the field being added to the database is a generated sequence number that uses Sybase's or SQL Server's native sequencing, use one of the `addIdentityField()` methods instead of the `addField()` method. The `addIdentityField()` methods have the following definitions:

```
addIdentityField(String fieldName, Class type)
addIdentityField(String fieldName, Class type, int fieldSize)
```

Example of table definition

The following example shows the table creation mechanism taken from the TopLink Employee Demo.

Example B–2 Creating a TableDefinition for the EMPLOYEE table

```
public static TableDefinition employeeTable()
{
TableDefinition definition;
definition = new TableDefinition();
definition.setName("EMPLOYEE");
definition.addIdentityField("EMP_ID", BigDecimal.class, 15);
definition.addField("VERSION", Integer.class);
definition.addField("F_NAME", String.class, 40);
definition.addField("L_NAME", String.class, 40);
definition.addField("START_DATE", java.sql.Date.class);
definition.addField("END_DATE", java.sql.Date.class);
definition.addField("START_TIME", java.sql.Time.class);
definition.addField("END_TIME", java.sql.Time.class);
definition.addField("GENDER",String.class,10);
definition.addField("ADDRESS_ID", BigDecimal.class, 15);
definition.addField("MANAGER_ID", BigDecimal.class, 15);
definition.addForeignKeyConstraint("employee_manager", "MANAGER_ID", "EMP_ID",
"EMPLOYEE");
definition.addForeignKeyConstraint("employee_address","ADDRESS_ID", "ADDRESS_
ID", "ADDRESS");
return definition;}
```

Creating tables on the database

Tables are created by passing the initialized `TableDefinition` object to the `DatabaseSession`'s schema manager.

TopLink provides two separate methods for creating tables.

- The `createObject()` method creates a new table on the database based on the table definition.

  ```
  SchemaManager schemaManager = new SchemaManager(session);
  schemaManager.createObject(Tables.employeeTable());
  ```

- The `replaceObject()` method destroys and recreates the schema entity on the database.

  ```
  schemaManager.replaceObject(Tables.addressTable());
  ```

Creating the sequence table

TopLink can automatically create a sequence table if required by the application. This can be done by calling the `createSequences()` method of the schema manager:

```
schemaManager.createSequences();
```

This configures the sequence table as defined in the session's `DatabaseLogin` and creates/inserts sequences for each sequence name of all registered descriptors in the session. If Oracle native sequencing is used, Oracle sequence objects are created.

Using the Schema Manager to manage Java and database type conversions

Table B–1 through Table B–5 list the field types that match a given class for each database supported by TopLink. This list is specific to the Schema Manager and does not apply to mappings (TopLink automatically performs conversion between any database types within mappings).

Table B–1 DB2 field types

Class	Data Type
java.lang.Boolean	SMALLINT
java.lang.Byte	SMALLINT
java.lang.Byte[]	BLOB
java.lang.Integer	INTEGER
java.lang.Float	FLOAT
java.lang.Long	INTEGER
java.lang.Double	FLOAT
java.lang.Short	SMALLINT
java.lang.String	VARCHAR
java.lang.Character	CHAR
java.lang.Character[]	CLOB
java.math.BigDecimal	DECIMAL
java.math.BigInteger	DECIMAL
java.sql.Date	DATE
java.sql.Time	TIME

Table B–1 DB2 field types (Cont.)

Class	Data Type
java.sql.Timestamp	TIMESTAMP

Table B–2 dBASE field types

Class	Data Type
java.lang.Boolean	NUMBER
java.lang.Byte	NUMBER
java.lang.Byte[]	BINARY
java.lang.Long	NUMBER
java.lang.Integer	NUMBER
java.lang.Float	NUMBER
java.lang.Double	NUMBER
java.lang.Short	NUMBER
java.lang.String	CHAR
java.lang.Character	CHAR
java.lang.Character[]	MEMO
java.math.BigDecimal	NUMBER
java.math.BigInteger	NUMBER
java.sql.Date	DATE
java.sql.Time	CHAR
java.sql.Timestamp	CHAR

Table B–3 Oracle field types

Class	Data Type
java.lang.Boolean	NUMBER
java.lang.Byte	NUMBER
java.lang.Byte[]	LONG RAW
java.lang.Integer	NUMBER

Table B–3 Oracle field types (Cont.)

Class	Data Type
java.lang.Long	NUMBER
java.lang.Float	NUMBER
java.lang.Double	NUMBER
java.lang.Short	NUMBER
java.lang.String	VARCHAR2
java.lang.Character	CHAR
java.lang.Character[]	LONG
java.math.BigDecimal	NUMBER
java.math.BigInteger	NUMBER
java.sql.Date	DATE
java.sql.Time	DATE
java.sql.Timestamp	DATE

Table B–4 Sybase field types

Class	Data Type
java.lang.Boolean	BIT default 0
java.lang.Byte	SMALLINT
java.lang.Byte[]	IMAGE
java.lang.Integer	INTEGER
java.lang.Long	NUMERIC
java.lang.Float	FLOAT(16)
java.lang.Double	FLOAT(32)
java.lang.Short	SMALLINT
java.lang.String	VARCHAR
java.lang.Character	CHAR
java.lang.Character[]	TEXT
java.math.BigDecimal	NUMERIC

Table B–4 Sybase field types (Cont.)

Class	Data Type
java.math.BigInteger	NUMERIC
java.sql.Date	DATETIME
java.sql.Time	DATETIME
java.sql.Timestamp	DATETIME

Table B–5 Microsoft Access field types

Class	Data Type
java.lang.Boolean	SHORT
java.lang.Byte	SHORT
java.lang.Byte[]	LONGBINARY
java.lang.Integer	LONG
java.lang.Long	DOUBLE
java.lang.Float	DOUBLE
java.lang.Double	DOUBLE
java.lang.Short	SHORT
java.lang.String	TEXT
java.lang.Character	TEXT
java.lang.Character[]	LONGTEXT
java.math.BigDecimal	DOUBLE
java.math.BigInteger	DOUBLE
java.sql.Date	DATETIME
java.sql.Time	DATETIME
java.sql.Timestamp	DATETIME

Session management services

TopLink provides statistics reporting and runtime configuration systems. There are two publicly-available APIs, `oracle.toplink.service.RuntimeServices` and `oracle.toplink.services.DevelopmentServices`.

RuntimeServices

The RuntimeServices API facilitates the monitoring of a running in-production system. It gives access to a number of statistical functions and reporting, as well as logging functions. Typical uses for RuntimeServices include turning logging on or off and generating real-time reports on the number and type of objects in a given cache or sub-cache.

For more information, see the topic on the RuntimeServices class in the JavaDoc for the TopLink Foundation Library API.

DevelopmentServices

The DevelopmentServices API enables developers to make potentially dangerous changes to a running non-production application. For example, DevelopmentServices API can be used to change the states of selected objects and modify and reinitialize identity maps.

This can be particularly useful for stress and performance testing pre-production applications, as well as offering the opportunity fast and easy prototyping.

For more information, see the topic on the RuntimeServices class in the JavaDoc for the TopLink Foundation Library API.

Using session management services

The session management service classes can be instantiated by passing a session to the constructor. After the service is instantiated, a graphical interface or an application can be attached to the object to provide statistical feedback and runtime option settings.

TopLink also ships with the session management services implemented as MBeans that can be deployed in applications and interfaced according to the MBean specification. TopLink does not currently provide any GUI interfaces into this API.

Example B–3 Implementing session management services as MBeans

```
import oracle.toplink.services.RuntimeServices;
```

```
import oracle.toplink.publicinterface.Session;
...
...
RuntimeServices service = newRuntimeServices ((session) session);
java.util.List classNames = service.getClassesInSession();
```

Session management services and BEA WebLogic Server TopLink support for BEA WebLogic Server automatically deploys the session management services to the JMX server. The JMX Mbeans can be retrieved with the following object names:

```
WebLogicObjectName("TopLink_Domain:Name=Development <Session><Name>
Type=Configuration");
WebLogicObjectName("TopLink_Domain:Name=Runtime <Session><Name>
Type=Reporting");
```

The `<Session><Name>` is the session and name under which the required session configuration stored in the `toplink-ejb-jar.xml` file.

The stored procedure generator

It is now possible to generate stored procedures based on the dynamic SQL generated for descriptors and mappings. After the stored procedures are generated, they can be *attached* to the mappings and descriptors of the domain object. In other words, the access to the database is through stored procedures and no longer through SQL.

> **Note:** Because of the nature of this feature, it has a number of limitations. It should be used only where database access is restricted to stored procedures, and not to enhance performance.

The stored procedure generator is split into two sections that are performed at development time. The first is the generation of the stored procedures, and the second is the attachment of the procedures to the descriptors and mappings.

Generation of stored procedures

Stored procedures can be generated for all descriptors and most relationship mappings.

There are two exceptions to this rule:

- Many to many mappings are not supported by the stored procedure generator.

- Stored procedures for Read operations are not generated for the Oracle platform.

As well as descriptors and mappings, stored procedures are also generated for updates and selects of sequence numbers.

To tell TopLink to use these stored procedures whenever actions are performed on this class, an amendment class is created. This class contains a method that attaches the stored procedures to each descriptor.

Example B–4 Stored procedures generated directly on the database

An amendment class called `com.demo.Tester` is created in the file `C:/temp/Tester.java`.

```
SchemaManager manager = new SchemaManager(session);
manager.outputDDLToDatabase();
manager.generateStoredProceduresAndAmendmentClass("C:/temp/",
"com.demo.Tester");
```

Example B–5 Generating stored procedures to a file

```
SchemaManager manager = new SchemaManager(session);
manager.outputDDLToFile("C:\Temp\test.sql");
manager.generateStoredProceduresAndAmendmentClass("C:/temp/",
"com.demo.Tester");
```

Attaching the stored procedures to the descriptors

After the stored procedures are created successfully on the database and the amendment file is created, the descriptors can be told to use these descriptors.

- Before logging in, call a method on the generated amendment class:

  ```
  Session session = project.createDatabaseSesssion();
  com.demo.Tester.amendDescriptors(project);
  ```

  This method sets up all the descriptors to use the procedures that were generated previously.

The Session Console

The session console is a tool to test descriptors, test logging in to the database, and view the performance of reading objects described by the descriptors from the database. Ideally, you should create a project using TopLink Mapping Workbench and use the session console to test the project file before doing further development.

Requirements

Make sure `toplink.jar`, `toplinksdk.jar`, `toplinksdkxerces.jar`, `xerces.jar`, and `tools.jar` are in your CLASSPATH (see *Oracle9iAS TopLink Getting Started*).

The session console is compatible with Swing (JFC) 1.02, Swing 1.03, and Java 2.

- If you are using JDK 1.1, you must install Swing and ensure that it is on your CLASSPATH correctly.
- The JDK 1.2 (Java 2) version of TopLink provides a version of the session console that is compatible with the JDK 1.2 version of Swing. Make sure you are using the JDK 1.2 version of TopLink when using JDK 1.2.

Using session console features

1. Start the session console.
   - If you are using IBM VisualAge for Java, run `oracle.toplink.tools.sessionconsole.SessionConsole` in the "TopLink Development Tools" project.

   OR

   - If you are using Windows or Windows NT, run Session Console in the "TopLink for Java Foundation Library" application group.

   OR

   - If you are using Windows or Windows NT, run the "SessionConsole.cmd" file from the *<INSTALL_DIR>*`\core` directory.

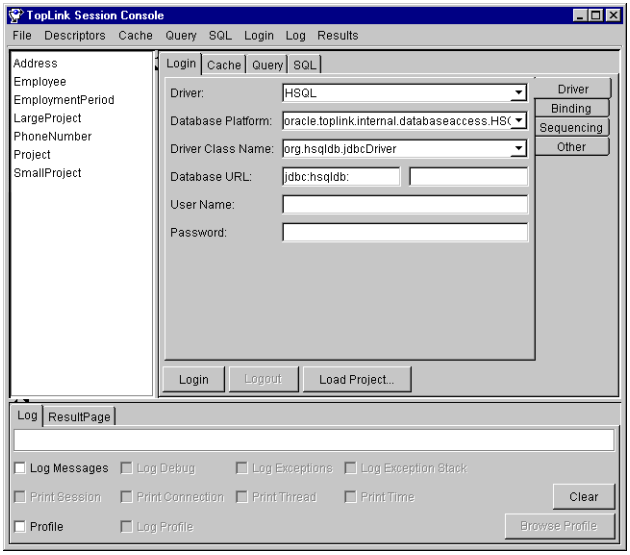
   OR

   - Enter the following text at the command line:

     <INSTALL_DIR>`\core\SessionConsole`

2. Select **File** > **Load Project**, then select the deployed XML project file you want to load (for example, `Employee.xml`).
3. Click the **Login** tab, fill in the Login information, and click the **Login** button.

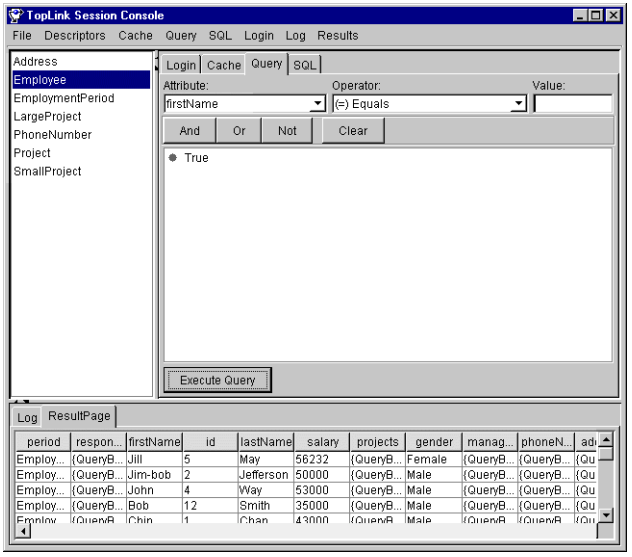
Figure B–1 TopLink Session Inspector—Login



4. To test reading all of the objects described by a descriptor from the database, select a descriptor and click the "Execute Query" button from the "Query" tab.

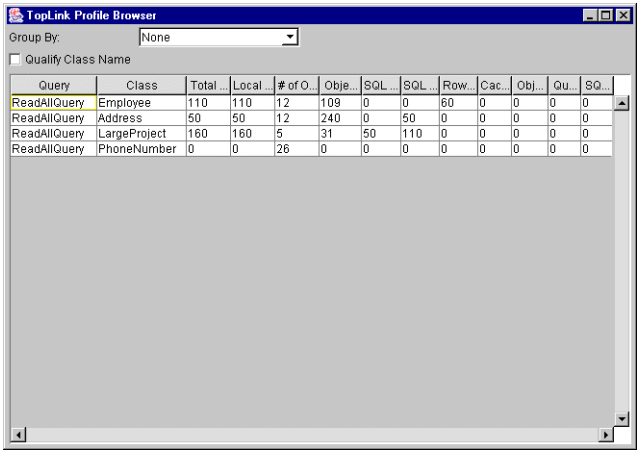
The session console executes the `ReadAllQuery` and displays the results in the Results pane.

Figure B–2 TopLink Session Inspector—Query



5. To view the performance of executing queries, check the **Profile** checkbox in the Log pane, and then repeat Step 4 for one or more descriptors.
6. Click the **Browse Profile** button in the Log pane to see the results. The total time is measured in milliseconds.

Figure B–3 TopLink Profile Browser



7. If you are using IBM VisualAge for Java, inspect the object that was read.
   1. Click the **Cache** tab.
   2. Select a descriptor that has objects displayed in the Cache pane.
   3. Select the object to be inspected and click the **Inspect (VA)** button.
8. If desired, execute SQL clauses in the SQL pane and view the result in the Result pane. Type the SQL clause, highlight this clause, and click either the Select button or the Update button.

   For example, an SQL clause to select all fields from Employee:

   ```
   SELECT * FROM EMPLOYEE
   ```

   Note: If your database contains important information, do not execute any SQL clause that modifies the database.

Launching the session console from code

The session console can also be launched from code. When debugging, you can launch the session console in your application. The session console can browse any of the TopLink sessions.

Example B–6 Launching the session console

```
SessionConsole.browse(toplinkSession);
```

The Performance Profiler

The performance profiler detects performance bottlenecks in a TopLink application. When it is enabled, the profiler logs a summary of the performance statistics for every query executed. The profiler also allows for a summary of all executed queries to be logged for a profile session.

The profiler currently logs the following information:

- The query type and arguments
- The total execution time of the query (in milliseconds)
- The total number of objects affected
- The number of objects handled per second
- The number of milliseconds per object

- The total time spent in the object cache
- The total time spent on preparing the SQL
- The total time spent on executing the SQL
- The total time spent on fetching rows

Using the profiler

To invoke the profiler from the database session, use the `setProfiler(new PerformanceProfiler())` method. To end a profiling session, use the `clearProfiler()` method.

The profiler is an instance of the `PerformanceProfiler` class, found in `oracle.toplink.tools.profiler` It can be accessed by calling the session's `getProfiler()` function.

The profiler supports the following public API:

- `logProfile()`: sets whether, after each query execution, the profile result should be logged; by default this is true
- `dontLogProfile()`: sets log profile to false
- `setShouldLogProfile(boolean value)`: sets log profile to 'passed' in boolean value
- `logProfileSummaryByClass()`: logs the profile summary by class
- `logProfileSummaryByQuery()`: logs the profile summary by query

Example B–7 The execution of a read query

```
session.setProfiler(new PerformanceProfiler());
Vector employees = session.readAllObjects(Employee.class);
```

The output generated by the profiler for the code is:

```
Begin Profile of{
ReadAllQuery(oracle.toplink.demos.employee.domain.Employee)Profile(ReadAllQuery,
,# of obj=12,time=1399,sql execute=217,prepare=495,row
fetch=390,time/obj=116,obj/sec=8)
} End Profile
```

Browsing the profiler results

The profiler results can be browsed graphically using the profile browser. The profile browser can be launched from code from your application. The profile browser is found in the `oracle.toplink.tools.sessionconsoleConsole` package.

Example B–8 Launching the profile browser

```
ProfileBrowser.browseProfiler(session.getProfiler());
```

C

TopLink Session Configuration File

The TopLink session configuration file, called `sessions.xml`, is a Java ResourceBundle that is read in using the locale. It may be necessary to rename the `sessions.xml` file to a locale specific name. For example `TopLink_en_US.properties` instead of `sessions.xml`.

Each TopLink project belongs to a TopLink session. To deploy beans that belong to different projects, add an appropriate TopLink session information section in the `sessions.xml` file as demonstrated in the Account Demo's sample

Contents of the sessions.xml file

```
# The XML Jar Location
XMLJarLocation = SPECIFY_JAR_LOCATION_HERE
# The TopLink session information for the beans in the Account Demo
session.YOUR_SESSION_NAME.projectClass = examples.ejb.cmp.account.AccountProject
session.YOUR_SESSION_NAME.platform =
oracle.toplink.internal.databaseaccess.OraclePlatform
session.YOUR_SESSION_NAME.profile = false
session.YOUR_SESSION_NAME.logProfile = false
session.YOUR_SESSION_NAME.logMessages = true
session.YOUR_SESSION_NAME.logDebug = true
session.YOUR_SESSION_NAME.logExceptions = true
session.YOUR_SESSION_NAME.useExternalConnectionPooling = true
session.YOUR_SESSION_NAME.useExternalTransactionController = true
session.YOUR_SESSION_NAME.externalTransactionController =
oracle.toplink.jts.was.WebSphereJTSExternalTransactionController
session.YOUR_SESSION_NAME.writePoolMax = 1
session.YOUR_SESSION_NAME.writePoolMin = 1
session.YOUR_SESSION_NAME.readPoolMax = 1
session.YOUR_SESSION_NAME.readPoolMin = 1
session.YOUR_SESSION_NAME.amendmentClass =
```

```
examples.ejb.cmp.order.SessionAmendment
session.YOUR_SESSION_NAME.amendmentMethod = configureSession
```

This code contains the following variables:

YOUR_SESSION_NAME The toplink_session_name environment variable set in the bean's deployment descriptor.

XMLJarLocation The property `XMLJarLocation` points to the directory in which your `.jar` files have been saved.

projectClass It is necessary to have a TopLink project class file. You can generate a project class file using the TopLink Session Console.

platform The database platform. This can also be specified in the TopLink project.

- For Oracle databases set this to `oracle.toplink.internal.databaseaccess.OraclePlatform`
- For DB2 databases set this to `oracle.toplink.internal.databaseaccess.DB2Platform`
- For Sybase databases set this to `oracle.toplink.internal.databaseaccess.SybasePlatform`
- For SQL Server databases set this to `oracle.toplink.internal.databaseaccess.SQLServerPlatform`

logProfile This flag indicates whether or not to use the profiler. The profiler can be used to log a summary of each query that is executed. This may be used during development but for optimized performance it is recommended that this be disabled during production.

logDebug Set debug messages logging. Debug messages will be dumped through TopLink to the session's log. By default this is System.out, but can be set to any Writer.

logExceptions Toggle exception logging.

useExternalConnectionPooling If set to true then a DataSource must be set for the bean. This DataSource is used for all database connections. If this is set to false then TopLink uses its own internal connection pooling.

useExternalTransactionController If set to true then TopLink database calls are synchronized with the container's "Transaction Manager".

externalTransactionController This is specific to your application server and coordinates with the appropriate JTS. For example, for IBM WebSphere set this to `oracle.toplink.jts.was.WebSphereJTSExternalTransactionController`

writePoolMax This is set only when an external transaction controller is not being used. It indicates the maximum number of write connections in the TopLink connection pool.

writePoolMin This is set only when an external transaction controller is not being used. It indicates the minimum/initial number of write connections in the TopLink connection pool.

readPoolMax This is set only when an external transaction controller is not being used. It indicates the maximum number of read connections in the TopLink connection pool.

readPoolMin This is set only when an external transaction controller is not being used. It indicates the minimum/initial number of read connections in the TopLink connection pool.

amendmentClass The class that contains the public class that amends the session.

amendmentMethod A public static method that takes a `oracle.toplink.threetier.ServerSession` as a parameter. This method is run when the TopLink session is created during bean deployment.

Converting from TOPLink.properties file to sessions.xml

The sessions.xml file is similar in function to the `TOPLink.properties` files from previous version of TopLink. If you are upgrading from a version of TopLinkhat used the `TOPLink.properties` file, the following table, which identifies settings in the `TopLink.properties` file and their equivalents in the `sessions.xml` file, will be of use to you.

Table C–1 TopLink.properties and sessions.xml equivalents

Element	TopLink.properties	sessions.xml
Session name		<toplink-configuration> <session> <name>YOUR_SESSION_NAME </name>
xml.jar location	XMLJarLocation=[INSTALL_ DIR}\TopLink\core\lib\xerces.jar	not required Use SessionManager and XMLLoader API to specify the parser JAR location and options such as classloader, etc.
Project class and session name	session.YOUR_SESSION_NAME .projectClass=examples.ejb .cmp.account.AccountProject	<session> <project-class> examples.ejb.cmp.account .AccountProject </project-class> </session>
Platform class	session.YOUR_SESSION_NAME .platform=TOPLink.Private .DatabaseAccess .OraclePlatform	<login> <platform-class> oracle.toplink.internal.databaseaccess .OraclePlatform </platform-class> **Note:** You will need to change the existing platform class name to the new oracle.toplink.* class name. This can be done by running the package rename tool on the TOPLink.properties file.
Profiling	session.YOUR_SESSION_NAME .profile=true session.YOUR_SESSION_NAME .logProfile=true	<session> <profiler-class> oracle.toplink .profiler.PerformanceProfiler</profiler -class>

Table C–1 TopLink.properties and sessions.xml equivalents (Cont.)

Element	TopLink.properties	sessions.xml
Message logging	session.YOUR_SESSION_NAME .logMessages=true session.YOUR_SESSION_NAME .logDebug=true session.YOUR_SESSION_NAME .logExceptions=true	<session> <log-messages>true </log-messages> <logging-options> <log-debug>true </log-debug> <log-exceptions>true </log-exceptions> </logging-options>
External connection pooling	session.YOUR_SESSION_ NAME.useExternalConnectionPooling=true	<login> <uses-external-connection-pool> true</uses-external-connection-pool>

Table C–1 TopLink.properties and sessions.xml equivalents (Cont.)

Element	TopLink.properties	sessions.xml
External transaction collector	session.YOUR_SESSION_NAME .useExternalTransactionController=true session.YOUR_SESSION_NAME .externalTransactionController=oracle.toplink.jts.was.WebSphereJTSExternalTransactionController	<session> <login> <uses-external-transaction-controller>true </uses-external-transaction-controller> </login> <external-transaction-controller-class> oracle.toplink.jts.was.WebSphereJTSExternalTransactionController </external-transaction-controller-class>
Connection pooling	session.YOUR_SESSION_NAME .writePoolMax=1 session.YOUR_SESSION_NAME .writePoolMin=1 session.YOUR_SESSION_NAME .readPoolMax=1 session.YOUR_SESSION_NAME. readPoolMin=1	<session> … <connection-pool> <is-read-connection-pool>true </is-read-connection-pool> <name>sampleReadConnectionPool</name> <min-connections>1 </min-connections> <max-connections>1 </max-connections> </connection-pool>
Amendment class and amendment method	session.YOUR_SESSION_ NAME.amendmentClass=examples.ejb .cmp.order.SessionAmendment session.YOUR_SESSION_ NAME.amendmentMethod= configureSession	These options are not present in the Sessions.xml for TopLink, as the Sessions.xml allows for a greater degree of control over session configuration. If you wish to customize your session in code the you may use the Session 'preLogin' event. A session event class can be specified in the Sessions.xml file to register a listener for the preLogin, and other session events. For more information on using this API, please consult your TopLink documentation.

D

EJBQL Syntax

Oracle TopLink supports the syntax for EJBQL as described in the Enterprise JavaBeans Specification v2.0. This section discusses the syntax for implementing EJBQL.

About Backus Naur Form

Backus Naur Form (BNF) is a formal notation used to describe the syntax of a given language. It includes the following elements:

Table D–1 BNF notation elements

Element	Use
::=	meaning "is defined as"
|	meaning "or"
< >	angle brackets used to contain the name of undefined elements. For example, consider the following code `<INSTALL_DIR>\TopLink\core\sessions_4_5.dtd` The angle brackets indicate that the text within is not defined by the code, but rather, requires the user to insert the appropriate string.
[ ]	optional elements
[ ]*	optional elements of which type there can be 0-*n*.
{ }+	elements, of which type there can be 1-*n*.

EJBQL language definition

The syntax for EJBQL is defined in the Enterprise JavaBeans Specification v2.0 The following is the EJBQL BNF extracted from the Enterprise JavaBeans Specification Version 2.0 dated July, 2001.

```
EJB QL ::= select_clause from_clause [where_clause]

from_clause ::= FROM identification_variable_declaration [, identification_
variable_declaration]*
    identification_variable_declaration ::=
        collection_member_declaration |
        range_variable_declaration
    collection_member_declaration ::=
        IN (collection_valued_path_expression) [AS ] identifier
    range_variable_declaration ::=
        abstract_schema_name [AS ] identifier
    single_valued_path_expression ::= {
        single_valued_navigation |
        identification_variable}.cmp_field |
        single_valued_navigation
    single_valued_navigation ::=
        identification_variable.[single_valued_cmr_
        field.]*
        single_valued_cmr_field
    collection_valued_path_expression ::=
        identification_variable.[single_valued_cmr_
        field.]*
        collection_valued_cmr_field

select_clause ::= SELECT [DISTINCT ] {
    single_valued_path_expression |
    OBJECT (identification_variable)}

where_clause ::= WHERE conditional_expression
    conditional_expression ::=
        conditional_term |
        conditional_expression OR conditional_term
    conditional_term ::=
        conditional_factor |
        conditional_term AND conditional_factor
conditional_factor ::=
        [NOT ] conditional_test
    conditional_test :: =
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
between_expression ::=
    arithmetic_expression [NOT] BETWEEN
        arithmetic_expression AND arithmetic_expression
in_expression ::=
    single_valued_path_expression [NOT] IN
        (string_literal [, string_literal]* )
like_expression ::=
    single_valued_path_expression [NOT] LIKE
    pattern_value [ESCAPE escape-character]
null_comparison_expression ::=
    single_valued_path_expression IS [NOT] NULL
empty_collection_comparison_expression ::=
    collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= {
        single_valued_navigation |
        identification_variable |
        input_parameter}
        [NOT] MEMBER [OF] collection_valued_path_expression
comparison_expression ::=
    string_value { =|<>} string_expression |
    boolean_value { =|<>} boolean_expression} |
    datetime_value { = | <> | > | < } datetime_expression |
    entity_bean_value { = | <> } entity_bean_expression |
    arithmetic_value comparison_operator single_value_designator
arithmetic_value ::=
    single_valued_path_expression |
    functions_returning_numerics
single_value_designator ::=
    scalar_expression
comparison_operator ::=
    = | > | >= | < | <= | <>
scalar_expression ::=
    arithmetic_expression
arithmetic_expression ::=
```

```
    arithmetic_term |
    arithmetic_expression { + | - } arithmetic_term
arithmetic_term ::=
    arithmetic_factor |
    arithmetic_term { * | / } arithmetic_factor
arithmetic_factor ::=
    { + |- } arithmetic_primary
arithmetic_primary ::=
    single_valued_path_expression |
    literal |
    (arithmetic_expression) |
    input_parameter |
    functions_returning_numerics
string_value ::=
    single_valued_path_expression |
    functions_returning_strings
string_expression ::=
    string_primary |
    input_expression
string_primary ::= single_valued_path_expression |
    literal |
    (string_expression) |
    functions_returning_strings
datetime_value ::=
    single_valued_path_expression
datetime_expression ::=
    datetime_value |
    input_parameter
boolean_value ::=
    single_valued_path_expression
boolean_expression ::=
    single_valued_path_expression |
    literal |
    input_parameter
entity_bean_value ::=
    single_valued_navigation |
    identification_variable
entity_bean_expression ::=
    entity_bean_value |
    input_parameter
functions_returning_strings ::=
    CONCAT (string_expression, string_expression) |
    SUBSTRING (string_expression, arithmetic_expression, arithmetic_
    expression)
functions_returning_numerics::=
```

```
LENGTH (string_expression) |
LOCATE (string_expression, string_expression[, arithmetic_expression]) |
ABS (arithmetic_expression) |
SQRT (arithmetic_expression)
```

Index

A

addConversionValue() method, 7-4
addDirectMapping() method, 7-2
addField() method, B-2
addFieldTransformation() method, 7-7
addForeignKeyConstraint() method, B-2
addIdentityField() method, B-2
addPrimaryKeyField() method, B-2
addPrimaryKeyFieldName, 8-1
addTableName method, 8-9
addToAttributeOnlyConversionValue() method, 7-5
aggregate collection mapping, 7-20
aggregate object mapping, 7-12
  example, 7-11
  in Java, 7-11
AggregateObjectMapping class, 7-11
AllFieldsLockingPolicy, 1-58, 1-59
application server, 2-5
array mapping, 7-27
array mappings
  about, 7-26
  implementing in Java, 7-27

B

batch reading, 6-3
batch writing, 1-17, 6-10
beans
  enterprise Java beans, 3-1
  entity bean model, 3-9
  session beans, 3-2
  stateful beans, 3-3
  stateless, 3-3
bi-directional relationship
  in one-to-one mappings, 7-13
binding, 1-16
binding and parameterized SQL
  binding string data, 1-16
  binding using parameters, 1-16
  binding using streams, 1-16
  explained, 1-16
boolean logic in expressions, 1-29
bridge
  JDBC-ODBC, 1-12
  other than Sun JDBC-ODBC, 1-13

C

cache
  internal query object cache, 1-81
  isolation, 1-18
  using identity maps, 1-2
cache identity map, 6-2
caching
  three-tier, 2-6
  using the readObject ( ) method, 1-26
cascading write queries
  compared to non-cascading, 1-82
ChangedFieldsLockingPolicy, 1-58, 1-59
class loader
  in Conversion Manager, 1-10
  resolving exceptions, 1-11
classes
  AggregateObjectMapping, 7-11
  CursoredStream, 1-92
    described, 1-92
    optimizing, 1-94

classes (cont.)
DatabaseException, 1-24
DatabaseLogin
creating the sequence table, B-5
described, 1-11
DatabaseSession
creating, 1-3
creating tables on database, B-4
described, 1-2
logging SQL and messages, 1-5
public methods, 1-8
session query operations, 1-22
DataModifyQuery
described, 1-72
DataReadQuery
described, 1-72
DeleteObjectQuery, 1-82
described, 1-72
DirectCollectionMapping, 7-16
DirectReadQuery
described, 1-72
DirectToFieldMapping, 7-2
Expression, 1-27
ExpressionBuilder, 1-31
InsertObjectQuery, 1-82, 1-83
and Unit of Work, 1-50
described, 1-72
ManyToManyMapping, 7-24
NestedTableMapping, 7-33
ObjectRelationalDescriptor, 8-19
ObjectTypeMapping, 7-4
OneToManyMapping, 7-23
OneToOneMapping, 7-13
OptimisticLockException, 1-59
PerformanceProfiler, B-16
ReadAllQuery
described, 1-72
ReadObjectlQuery
described, 1-72
ReadObjectQuery, 1-75
ReportQuery
described, 1-72
ScrollableCursor, 1-92
SerializedObjectMapping, 7-6
TableDefinition, B-2
classes (cont.)
TransformationMapping, 7-7
TypeConversionMapping, 7-3
UnitOfWork, 1-40
using to modify databases, 1-23
UpdateObjectQuery, 1-82
described, 1-72
example, 1-82
ValueReadQuery
described, 1-72
VariableOneToOneMapping, 7-15
WriteObjectQuery
described, 1-72
clearProfiler() method, B-16
client sessions, 2-3, 2-5
collection class, 1-76
composite primary key, 7-14
concurrency, 2-9
connection policies, 2-12
connection pooling
described, 2-10
ServerSession, 2-11
container-managed persistent entity beans, 3-10
Conversion Manager
assigning a custom Conversion Manager to a session, 1-10
assigning a custom Conversion Manager to all subsequent sessions, 1-10
class loader, 1-10
described, 1-9
using, 1-9
using custom types, 1-10
copy policy
implementing in Java, 8-9
CORBA
message optimization, 6-15
TopLink support for, 2-2
Toplink transport layer support, 2-17
createObject() method, B-4
cursored streams
described, 1-92
example, 1-93
optimizing, 1-94
ReadAllQuery methods, 1-76
usage example, 2-19

CursoredStream class, 1-92
cursors, scrollable
  traversing, 1-95
  using, 1-95
custom query objects
  creating, 1-84
custom SQL, 1-24
custom SQL queries
  in TopLink query framework, 1-24
custom types
  assigning to a TopLink session, 1-10
custom types, using with Conversion Manager, 1-10

D

data optimization, 1-17
database access
  non-relational, 2-3
  using stored procedures, B-10
database and Java type conversion tables, B-5
database exceptions, 1-24
database login, 1-11
  example, 1-20
database sessions, defined, 1-2
database, logging out, 1-4
DatabaseException class, 1-24
DatabaseLogin class, using to store login information, 1-11
DatabaseRow, 7-7
DatabaseSession class
  creating tables on a database, B-4
  described, 1-2
  instantiating, 1-3
  logging SQL and messages, 1-5
  public methods, 1-8
  session queries, 1-22
data-level query
  example, 1-37, 1-97
DataModifyQuery, 1-72
DataReadQuery, 1-72
DataSources, using JDBC2.0, 1-19
DB2 field types, B-5
dBASE field types, B-6
delete operation, 1-41
DeleteObjectQuery
  defined, 1-72
  example, 1-82
development tools
  profiler
    described, B-1
    using, B-15
  schema manager
    described, B-1
    using, B-2
  session console
    described, B-1
    using, B-11
direct collection mapping, 7-17
direct collection mappings
  example, 7-17
  in Java code, 7-16
direct connect drivers, 1-19
direct map mapping, 7-22
DirectCollectionMapping class, 7-16
DirectReadQuery, 1-72
direct-to-field mappings, 7-3
  in Java code, 7-2
DirectToFieldMapping class, 7-2
Distributed Transaction Processing (DTP), 2-26
does exist write object, 6-11
drivers, direct connect, 1-19
DTP see Distributed Transaction Processing, 2-26

E

EJB Entity Beans, 2-3
EJB see Enterprise Java Beans, 3-1
EJB Session Beans, 2-2, 2-17
enterprise applications, 2-1
Enterprise Java Beans (EJB), 3-1
enterprise Java beans, in TopLink for BEA WebLogic, 3-1
entity bean model, 3-9
entity beans
  container managed, 3-10
  features, 2-3
event
  implementing in Java, 8-21
event manager, 1-68

events
listeners, 8-22
registering listeners, 8-22
events, session, 1-68
examples
identity maps, 8-17
multiple tables, 8-14
optimistic locking, 1-60
performance optimization, 6-6, 6-8
read query, B-17
report query, 1-89
scrollable cursors, 1-95
serialized object mapping, 7-6
session broker, 2-21
session event manager, 1-70
SQL queries, 1-96
stored procedure call, 1-98
stored procedures, generating, B-11
transformation mapping, 7-8
type conversion mapping, 7-3
unit of work, 1-45, 1-48, 1-54
variable one-to-one mapping, 7-15, 7-16
write, write all, 1-42
exception handlers, 1-6
exceptions
database, 1-24
Expression class, 1-27
expression components, 1-28
ExpressionBuilder, 1-31
expressions
parameterized, 1-33
using Boolean logic, 1-29

F

field locking policies, 1-57, 1-59, 8-16
field types
DB2, B-5
dBASE, B-6
Microsoft Access, B-8
Oracle, B-6
Sybase, B-7
fields, adding to a table definition, B-3
foreign keys, 1-34
addForeignKeyConstraint(), B-2
direct collection mappings, 7-17
one-to-one mappings, 7-13
framework, query, 1-22
full identity map, 6-2

G

getInheritancePolicy(), 8-2
getWrapperPolicy(), 8-21

H

HTML, 2-2

I

identity maps, 1-2, 1-4, 1-77
cache identity map, 6-2
example, 1-77
full identity map, 6-2
implementing in Java, 8-17
soft cache identity map, 6-2
soft cache weak identity map, 6-2
weak identity map, 6-2
indirection, 2-18, 6-2
implementing in Java, 8-18
in transformation mapping, 7-8
example, 7-8
Java class requirements, 8-20
one-to-many mappings, 7-13
specifying for classes in Java, 8-20
Informix
using native sequencing, 1-15
inheritance
class extraction methods, 8-3
creating hierarchy in Java, 8-2
implementing in Java, 8-2
leaf classes, 1-25
querying on hierarchy, 1-25
transformed to relational model, 6-18
InheritancePolicy method, 8-8
in-memory query, 1-78
insert operation, 1-40, 1-41

InsertObjectQuery, 1-72
instantiation policy
  implementing in Java, 8-15
  methods, 8-15
  overridinging in Java, 8-15
integrity checker, 1-6
interfaces
  implementing in Java, 8-8
internal query object cache, 1-81
isolation, cache, 1-18
OTS see, 2-28
Iterator interface, 1-94
iterator interface, 1-92

J

Java and database type conversion tables, B-5
Java Messaging Service
  described, 2-58
  setting up in Java, 2-59
  setting up in session configuration file, 2-59
Java optimization, 6-14
Java streams, 1-92
Java Transaction Service (JTS), 2-3, 2-25, 2-28
  example, 2-25
JavaBeans, 3-1
JConnect (Sybase), 1-18
JDBC2.0 DataSources, 1-19
JDBC-ODBC bridge, 1-12
JMS, see Java Messaging Service
joining, 6-3
joins, outer, 1-37
JPS, 2-2
JTS (Java Transaction Service)
  TopLink feature support, 2-3
  TopLink integration, 2-25
JTS implementation
  extending in Toplink, 2-34
  in BEA WebLogic, 2-34

K

keys
  foreign, 7-13, B-2
  primary, B-2
  primary, composite, 7-14

L

leaf classes, 1-25
listeners, event, 8-22
ListIterator interface, 1-92
locking
  pessimistic, 1-64
locking policies
  implementing in Java, 8-16
logging into the database, 1-11
logging out, 1-4
login class
  creating for projects created in Mapping Workbench, 1-12
  creating for projects not created in Mapping Workbench, 1-11
login parameters
  setting in code, 1-13

M

manager, session events, 1-68
manual transactions, 1-18
many-to-many mapping, 7-25
  example, 7-25
  Java, 7-24
ManyToManyMapping class, 7-24
mapping
  attribute, 7-12
  direct-to-field, 7-3
  many-to-many, 7-25
  object type, 7-5
  relationship, 7-12
  serialized object, 7-6
  transformation, 7-8
  transformation, properties, 7-10
  type conversion, 7-3, 7-4

Mapping Workbench
multiple projects, 2-23
sessions, registering, 1-3
methods
addDirectMapping(), 7-2
addField(), B-2
addForeignKeyConstraint(), B-2
addIdentityField(), B-2
addPrimaryKeyField(), B-2
addTableName, 8-9
addToAttributeOnlyConversionValue(), 7-5
clearProfiler(), B-16, B-17
copy policy, 8-9
createObject(), B-4
instantiation, 8-15
replaceObject(), B-4
setDefaultAttributeValue(), 7-5
setName(), B-2
setProfiler(), B-16, B-17
wrapper policy, 8-21
Microsoft Access field types, B-8
multiple tables
implementing in Java, 8-9
implementing in Java when primary keys are named differently, 8-10
implementing in Java when primary keys match, 8-9
implementing in Java when related by foreign key, 8-11
implementing in Java, non-standard table relationships, 8-12
multi-processing, 6-13

N

native sequencing, 1-15
Oracle, B-5
SQL Server, B-3
Sybase, B-3
nested table mapping, 7-34
nested table mappings
about, 7-33
Java, 7-33
NestedTableMapping class, 7-33
non-cascading write queries
compared to cascading, 1-82
creating using dontCascadeParts ( ) method, 1-82
non-relational database access, 2-3
non-standard table relationships
implementing in Java, 8-12
Normally, 8-3

O

object array mapping, 7-29
about, 7-28
object array mappings
implementing in Java, 7-28
object identity, 1-2, 1-4
object indirection, 6-2
object model, 1-25, 1-27, 6-15
object reading, partial, 6-3
Object Transaction Service, 2-28
Object Transaction Service (OTS), 2-28
object type mapping, 7-5
example, 7-5
object, cache, 1-58, 1-81
object-relational descriptors
implementing in Java, 8-19
ObjectRelationalDescriptor class, 8-19
objects
query, 1-71, 1-85
ObjectTypeMapping class, 7-4
one-to-many mapping, 7-24
example, 7-23
Java, 7-23
OneToManyMapping class, 7-23
one-to-one mapping, 7-14
example, 7-13
Java, 7-13
variable class relationships, 7-16
OneToOneMapping class, 7-13
operators
boolean logic, 1-29
optimistic locking, 1-57
database exception, 1-24
field locking policy, 1-58
version locking policy, 1-58

OptimisticLockException class, 1-59
optimization
  data, 1-17
  Java, 6-14
  performance, 6-1
  schema, 6-15
Oracle
  field types, B-6
  remote session support for, 2-17
  using native sequencing, 1-15
OrbixWeb, 2-17
outer joins, 1-37

P

parameter binding, 1-16
parameterized expressions
  described, 1-33
  example, 1-34
parameterized SQL
  described, 1-16
  enabling on queries, 1-75
  TopLink optimization features, 6-10
partial object reading, 6-3
Performace Profiler, B-15
performance optimization
  described, 6-1
  examples, 6-3
  using Performance Profiler, B-15
PerformanceProfiler class, B-16
persistent entity beans, 3-10
pessimistic locking
  described, 1-64
  example, 1-65
pooling, connection, 2-10
primary key
  addPrimaryKeyField(), B-2
  composite, 7-14
  implementing in Java, 8-1
Profiler, 1-5
profiler development tool, B-1, B-15
proxy indirection
  implementing in Java, 8-19

Q

queries
  cascading, 1-82
  SQL, 1-96
query
  in-memory, 1-78
  report, 6-3
query by example, 1-85
query framework, 1-22
query keys
  implementing in Java, 8-17
query methods, 1-25
query objects
  creating, overview, 1-73
  DataModifyQuery
    described, 1-72
  DataReadQuery
    described, 1-72
  defined, 1-81
  DeleteObjectQuery
    described, 1-72
  DirectReadQuery
    described, 1-72
  examples, 1-73, 1-74
  in TopLink query framework, 1-23
  InsertObjectQuery
    described, 1-72
  ReadAllQuery
    described, 1-72
  ReadObjectQuery
    described, 1-72
  relationship to database, 1-4
  ReportQuery
    described, 1-72
  UpdateObjectQuery
    described, 1-72
  using, 1-71
  using in place of session methods, 1-82
  ValueReadQuery
    described, 1-72
  WriteObjectQuery
    described, 1-72
query timeout example, 1-74
query, report, 1-89

R

read all operation, 1-26
read operation, 1-26
read query example, B-16, B-16, B-17
ReadAllQuery, 1-72
reading, batch, 6-3
readObject()
  example, 1-27
reference mapping, 7-32
  example, 7-32
  in Java, 7-32
ReferenceMapping class, 7-31
refresh operation, 1-27
relational mappings
  about, 7-26
relationship
  bi-directional, 7-13
  variable class, 7-16
remote connection using RMI
  example, 2-19
remote session, 2-13
replaceObject() method, B-4
report query
  use case, 6-3
  using, 1-89
ReportQuery, 1-72
RMI
  message optimization, 6-15
  remote session support, 2-17
  TopLink features, 2-2

S

samples. see examples
schema creation, 2-5
schema manager, B-5
schema manager development tool, B-1
schema, optimization, 6-15
scrollable cursors
  and cursored streams, 1-92
  traversing, 1-95
  using, 1-95
  using for ReadAllQuery, 1-76
ScrollableCursor class, 1-92
SDK for non-relational database access, 2-3
SelectedFieldsLockingPolicy, 1-58, 1-59
sequence numbers
  implementing in Java, 8-14
  preallocation, 6-12
  specifying, 1-14
  SQL Server, B-3
  Sybase, B-3
  write optimization features, 6-10
sequence table, 1-15
serialized object mappings, 7-6
  example, 7-6
  Java, 7-6
SerializedObjectMapping class, 7-6
server layer, 2-17
server sessions
  described, 2-3
  overview of use, 2-6
ServerSession connection options, 2-11
servlets, 2-2
Session, 7-7
session bean model, 3-2
session beans
  model, 3-2
  remote session support for, 2-17
  TopLink EJB features, 2-2
session broker, 2-20
session console development tool, B-1, B-11
session event manager, 1-68
session queries, 1-22
  in TopLink query framework, 1-22
session, remote, 2-13
SessionManager
  retrieving a session, 3-6
  session location, 3-6
sessions, database, 1-1, 1-2
sessions, logging out, 1-4
setAttributeClassification(), 7-3
setAttributeName(), 7-3, 7-4, 7-6, 7-7
setAttributeTransformation(), 7-7
setDefaultAttributeValue(), 7-5
setFieldClassification(), 7-3
setFieldName(), 7-2, 7-3, 7-4, 7-6
setGetMethodName(), 7-2, 7-3, 7-4, 7-6, 7-7
setName() method, B-2

setPrimaryKeyFieldName, 8-1
setProfiler() method, B-16, B-17
setSequenceNumberFieldName, 8-14
setSetMethodName(), 7-2, 7-3, 7-4, 7-6, 7-7
setWrapperPolicy(), 8-21
soft cache weak identity map, 6-2
SQL, 1-14, 1-27
  binding and parameterizing, 1-16
  custom, 1-24
  parameterized, 6-10
  queries, 1-96
SQL DISTINCT, 6-8
SQL queries
  in TopLink query framework, 1-24
SQL Server
  native sequencing, 1-15
SQL, parameterized, 1-75
stateful and stateless beans compared, 3-3
stateful beans, 3-3
stateful three-tier model, 2-5
stateless and stateful beans compared, 3-3
stateless beans, 3-3
stateless three-tier model, 2-5
stored procedures, 1-97
stored procedures, generating, B-10
streams, cursored, 1-94, 2-19
streams, Java, 1-92
structure mapping, 7-31
structure mappings
  Java, 7-30
StructureMapping class, 7-29
Swing, B-12
Sybase
  field types, B-7
  JConnect2.x, 1-18
  using native sequencing, 1-15

T

table definition, adding fields, B-3
TableDefinition
  creating for the EMPLOYEE table, B-4
TableDefinition class, B-2
TableDefinition, creating, B-4
tables
  creator/qualifier, 1-14
The Open Group, 2-26
three box model, 2-26
three-tier applications, 2-1
  migrating to scalable architecture, 1-2
TimestampLockingPolicy, 1-58
transactions, 1-42, 2-5
transactions, manual, 1-18
transformation mappings
  example, 7-8
  indirection, 7-8
  properties, 7-10
TransformationMapping class, 7-7
transport layer, 2-16
troubleshooting, unit of work, 1-54
two-phase commit with presumed rollback, 2-27
two-phase/two-stage commits, 2-21
type conversion mappings, 7-4
  example, 7-3
TypeConversionMapping class, 7-3

U

unit of work, 1-3, 1-42, 1-44, 1-46, 1-47, 1-48, 1-49, 1-50, 1-51, 2-5, 6-2, 6-10
  troubleshooting, 1-54
  validation, 1-53
unit of work, remote sessions, 2-19
unit of work, three-tier, 2-8
UnitOfWork class, 1-23, 1-40
update operation, 1-40, 1-41
UpdateObjectQuery, 1-72
useAllFieldsLocking, 1-59
useAllFieldsLocking, 8-16
useChangedFieldsLocking, 1-59
useChangedFieldsLocking, 8-16
useCloneCopyPolicy(), 8-9
useCloneCopyPolicy(String), 8-9
useConstructorCopyPolicy(), 8-9
useDefaultConstructorInstantiationPolicy(), 8-15
useFactoryInstantiationPolicy(), 8-15
useMethodInstantiationPolicy(), 8-15
useProxyIndirection(), 8-19
useSelectedFieldsLocking, 1-59

useSelectedFieldsLocking, 8-16
useTimestampLocking, 1-59
useTimestampLocking, 8-16
useVersionLocking, 1-59
useVersionLocking, 8-16

V

validation, 1-53
ValueReadQuery, 1-72
variable class relationships, 7-16
variable one-to-one mapping, 7-16
VariableOneToOneMapping class, 7-15
version fields, 1-57, 1-58
version locking policies, 1-57, 1-58, 1-59, 8-16
VersionLockingPolicy, 1-58
VisiBroker, 2-17

W

weak identity map, 6-2
weak identity map, soft cache, 6-2
WebLogic, 2-17, 2-31, 2-34, 3-1
wrapper policy
  implementing in Java, 8-21
  setting in Java, 8-21
write all operation, 1-41
write query objects, 1-82
WriteObjectQuery, 1-72
writing, batch, 1-17, 6-10

X

X/Open (The Open Group), 2-26