Programming Persistence

Forte™ for Java™ Programming Series

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 U.S.A.
650-960-1300

Part No. 816-1411-10
August 2001, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Contents

Figures

Tables

Preface

Welcome to the *Programming Persistence* book of the Forte™ for Java™ Programming Series. This book focuses on programming with persistent data—data stored in a database or other data store that is external to your applications. The book discusses the different persistence programming models supported by Forte for Java. It focuses on the Transparent Persistence technology provided by the Forte for Java integrated development environment (IDE).

This book is written for programmers who want to learn how to use the persistence programming models supported by Forte for Java. The book assumes a general knowledge of Java and database access technology. Before reading it, you should be familiar with the following subjects:

- Java programming language
- Relational database concepts (such as tables and keys)
- How to use the chosen database

You can create the examples in this book on the following platforms and operating systems:

- Solaris™ 8 *SPARC™ Platform Edition*
- Microsoft Windows 2000, SP2
- Microsoft Windows NT 4.0, SP6
- Red Hat Linux 6.2

All screen shots in this book are from the Windows NT version of the Forte for Java software. You should have no trouble translating the slight visual differences to other platforms. Although almost all procedures use the Forte for Java user interface, occasionally you might be instructed to enter a command at the command line. In such cases, examples are given with the prompt and syntax for a Microsoft Windows command window. For example:

```
c:\>cd MyWorkDir\MyPackage
```

To translate for UNIX® or Linux environments, simply change the prompt and use forward slashes:

```
% cd MyWorkDir/MyPackage
```

Before You Read This Book

This book is written for programmers who want to learn how to use the persistence programming models supported by Forte for Java. The book assumes a general knowledge of Java and database access technology. Before reading it, you should be familiar with the following subjects:

- Java programming language
- Relational database concepts (such as tables and keys)
- How to use the chosen database

How This Book Is Organized

The following briefly describes the contents of each chapter:

Chapter 1 explains what persistence is and establishes a framework for more detailed descriptions of Forte for Java persistence support in succeeding chapters. It also introduces a number of persistence programming models supported by the Forte for Java IDE.

Chapter 2 describes JDBC™ productivity enhancement tools provided by Forte for Java. These automate many JDBC programming tasks in building client components or applications that interact with a database.

Chapter 3 provides a brief overview to the Transparent Persistence programming model.

Chapter 4 describes the Transparent Persistence mapping tool and how to create a mapping between a set of Java programming language classes and a relational database.

Chapter 5 describes the Transparent Persistence runtime environment and illustrates how to use it to perform persistence operations. It also addresses various Transparent Persistence programming issues.

Chapter 6 describes the process for using persistence-capable classes with Enterprise Java Beans, the J2EE Reference Implementation, and the iPlanet Application Server.

Appendix A documents the system requirements necessary to use Transparent Persistence with the Forte for Java IDE.

Appendix B documents two JSP™ tags that perform Transparent Persistence functions.

Appendix C details unsupported features, areas where specific databases behave uniquely with Transparent Persistence, and file migration information for developers who have created classes using previous versions of Transparent Persistence

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Related Documentation

Forte for Java documentation includes books delivered in Acrobat Reader (PDF) format, online help, Readme files of example applications, and Javadoc™ documentation.

Documentation Available Online

The documents in this section are available from the Forte for Java portal, the `docs.sun.com`SM web site, and from Fatbrain.com, an Internet professional bookstore.

The documentation link of the Forte for Java portal is at `http://www.sun.com/forte/ffj/documentation/index.html`. The `docs.sun.com`SM web site is at `http://docs.sun.com`. Fatbrain.com is at `http://www.fatbrain.com/documentation/sun`.

- Release Notes (PDF format)

  Available for each Forte for Java edition. Describe last-minute release changes and technical notes.

- *Getting Started Guide* (PDF format)

  Available for each Forte for Java edition. Describes how to install Forte for Java on each supported platform and other pertinent information, including system requirements, command-line switches for starting the IDE, installed subdirectories, how to mount a JAR or zip file as a Javadoc filesystem in the IDE, and how to delete a project from the IDE.

- The Forte for Java Programming Series (PDF format)

  This series provides in-depth information on how to use various Forte for Java features to develop well-formed J2EE applications.

  - *Building Web Components* - part no. 816-1410-10

    Describes how to build a web application as a J2EE web module using JSP pages, servlets, tag libraries, and supporting classes and files.

  - *Programming Persistence* - part no. 816-1411-10

    Describes support for different persistence programming models provided by Forte for Java: JDBC and Transparent Persistence.

  - *Building Enterprise JavaBeans Components* - part no. 816-1401-10

    Describes how to build Enterprise JavaBeans components—session beans and entity beans with container-managed or bean-managed persistence—using the Forte for Java EJB Builder wizards and other graphical user interfaces.

  - *Building Web Services* - part no. 816-1400-10

    Describes how to use the tools provided by the Web Services module to build web services. Web Services are application business services published as Extensible Markup Language (XML) documents delivered over HTTP connections.

  - *Building JSP Pages That Use XML Data Services* - part no. 816-1399-10

    Describes how to use the Forte for Java Enterprise Service Presentation Toolkit to incorporate dynamic XML data in HTML.

- *Assembling and Executing J2EE Modules and Applications* - part no. 816-1402-10

  Describes how to assemble EJB modules and web modules into a J2EE application, and how to deploy and run a J2EE application.

- Forte for Java tutorials (PDF format)

  You can also find the completed tutorial applications in your user settings directory, under `sampledir/tutorial`.

  - *Forte for Java, Community Edition Tutorial* - part no. 816-1408-10

    Provides step-by-step instructions for building a simple J2EE web application using Forte for Java, Community Edition tools.

  - *Forte for Java, Enterprise Edition Tutorial* - part no. 816-1409-10

    Provides step-by-step instructions for building an application using Enterprise JavaBeans components, the test application facility, and the Forte for Java Web Services technology.

Online Help

Online help is available inside the Forte for Java development environment. You can access it by pressing the help key (Help on Solaris, F1 on Microsoft Windows and Linux), or by choosing Help > Contents. Either action displays a list of help topics and a search facility.

Examples

Several examples, with accompanying `Readme` files, that illustrate a particular Forte for Java feature are available in the `sampledir/examples` subdirectory of your user settings directory. In addition, you can download Enterprise Edition-specific examples from the Forte for Java portal and unzip them into the `examples` directory. Completed tutorial applications—including the applications described in *Forte for Java, Community Edition Tutorial* and *Forte for Java, Enterprise Edition Tutorial*—are in the `sampledir/tutorial` directory.

Javadoc Documentation

Javadoc documentation is available within the IDE for many Forte for Java modules. Refer to the release notes for instructions on installing this documentation. When you start the IDE, you can access this Javadoc documentation within the Javadoc pane of the Explorer.

Accessing Sun Documentation Online

A broad selection of Sun system documentation is located at:

`http://www.sun.com/products-n-solutions/hardware/docs`

A complete set of Solaris documentation and many other titles are located at:

`http://docs.sun.com`

Ordering Sun Documentation

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at:

`http://www.fatbrain.com/documentation/sun`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at:

`docfeedback@sun.com`

Please include the part number (816-1411-10) of your document in the subject line of your email.

CHAPTER 1

Overview of Persistence Programming

This chapter describes persistence and establishes a framework for more detailed discussions of Forte for Java persistence support in succeeding chapters. It also introduces a number of persistence programming models supported by Forte for Java.

About Persistence

A key aspect of most business applications is the programmatic manipulation of *persistent data*—long-lived data stored outside of an application. Although persistent data is read into transient memory for the purpose of using or modifying it, it is written out to a relational database or flat file system for long-term storage.

Representation of Persistent Data

In object-oriented programming systems, persistent data is represented in memory as one or more data objects manipulated by application code. In general, the correspondence between persistent data in a data store and its representation as a persistent data object in memory is achieved through a number of software layers as shown in FIGURE 1-1.

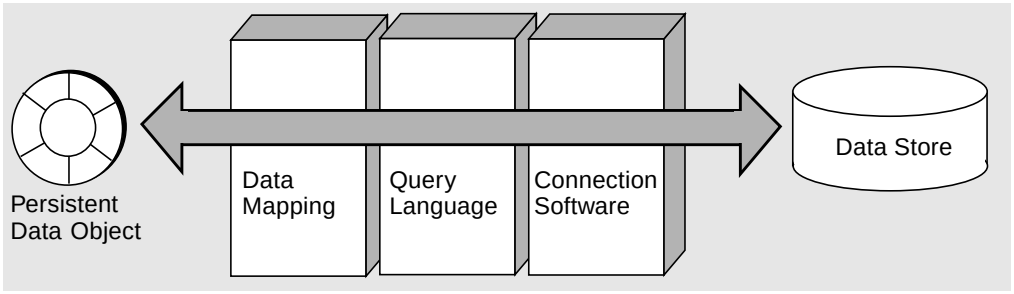

FIGURE 1-1 Basic Persistence Scheme

Each data store has an interface to the outside world through driver software used to set up and maintain a connection between the data store and an application. With this connection established, a query language is used to retrieve information in the data store and read it into an application, or conversely, to write data from the application into the data store. Another layer provides a mapping between data objects in memory and the information in the data store.

Through this general scheme, programmers can represent persistent data as runtime objects to be used and manipulated by an application. The scheme supports all basic persistence operations—often abbreviated as CRUD:

- Creating persistent data (inserting in a data store)
- Retrieving persistent data (selecting from a data store)
- Updating persistent data
- Deleting persistent data.

Application Issues

When programming applications, this relationship between data objects in memory and information in a data store is complicated by a number of issues. These include *synchronization*, *concurrency*, and *connection resources*.

- Synchronization

  An application needs to ensure that the two representations of data (in memory and in the data store) are kept synchronized. Any change to a persistent data object, for example, should take place only if that change also takes place in the data store. Since failure might occur in the process of writing to the data store, these changes need to be part of a single *transaction*. A transaction is a series of operations that commits only if all the individual operations are successful. If failure occurs, all changes need to be rolled back to their original state.

- Concurrency

  An application needs to provide for two or more users to have concurrent access to persistent data, and to ensure that the data not be corrupted. In other words, changes in the data made by any one user are known by other users in a timely fashion.

- Connection resources

  As the number of users of an application increases, the resources required to create and maintain large numbers of connections to a data store can become prohibitive. It is much more efficient to share or recycle these resources using a connection management and pooling scheme.

Synchronization, concurrency, and connection resources become increasingly important as the scale and complexity of an application increases. In an application in which a small number of clients are accessing a single database on a single computer, synchronization, concurrency, and connection resource requirements are easy to fulfill. However, as the number of clients, databases, and transactions grows, these issues can present a daunting programming challenge.

Java Database Programming Models

In the Java development environment, certain aspects of the interaction between persistent data objects and data stores have been standardized. Most database vendors provide drivers that interface with the Java execution environment (the Java Virtual Machine), and a standardized query language (SQL) that is generally used to perform persistence (CRUD) operations.

However, within this standardization, a number of models are available to support the programming of persistence operations, each corresponding to a specific persistence API. Forte for Java supports the following programming models:

- Java Database Connectivity (JDBC)
- Transparent Persistence

These different programming models will be described briefly in the following sections.

Java Database Connectivity (JDBC)

Java provides a standard persistence programming model, the JDBC API, to facilitate the coding of persistence operations. The JDBC API is a set of Java interfaces that you can use to perform basic persistence operations. Forte for Java provides JDBC tools and programming features based on the JDBC API, described in Chapter 2.

JDBC Programming Model

The JDBC programming model follows closely the software layers identified in FIGURE 1-1. You create a class to represent persistent data by writing code that maps fields of the class to columns and data types of one or more tables in a database system. You can then create an instance of that class (a persistent data object) and populate its fields with corresponding values from the database, or create a new instance, populate its fields, and write the data into the database.

FIGURE 1-2 illustrates the runtime objects involved in JDBC persistence operations. These objects are instances of classes that implement interfaces in the JDBC API. These objects are referenced by code in a persistence-aware component, also shown in FIGURE 1-2, that performs persistence operations.

For example, to read data into a persistent data object:

- Obtain a Connection to the database from a DriverManager object.
- Obtain a Statement from the Connection object.
- Pass to the Statement an SQL string representing a select query.

  The Statement is executed across the Connection, returning a ResultSet from the database.
- Extract data values from the ResultSet to populate the fields of your persistent data object.

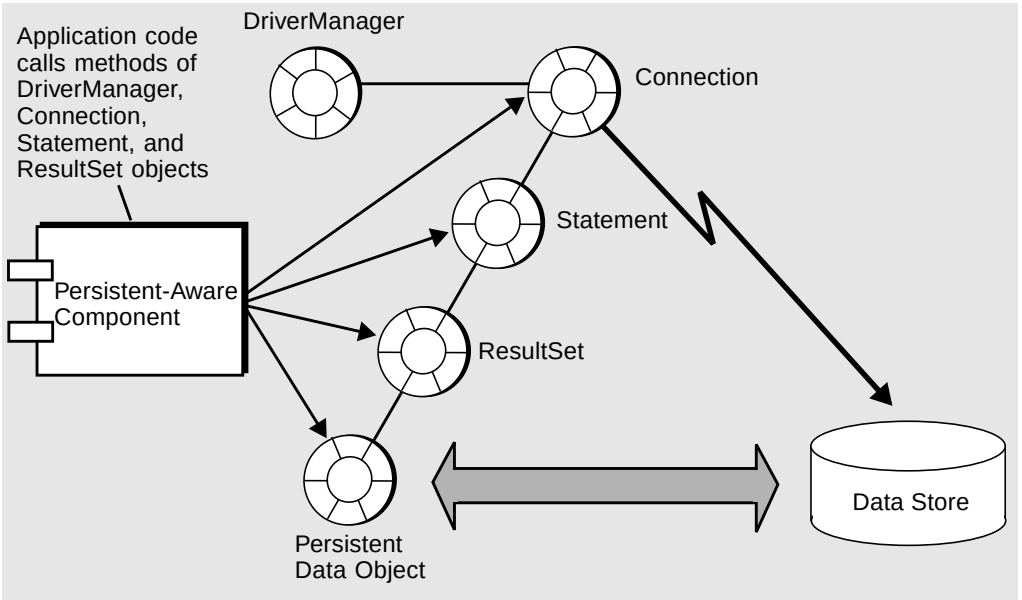

FIGURE 1-2 JDBC Programming Model

Similarly, you can write values from the persistent data object into the database using an SQL update statement. When you are finished with a statement or a connection, you close it using a method provided in the JDBC API.

JDBC compliant drivers are multi threaded; they support multiple concurrent connections. JDBC connections, in turn, support multiple statements executing concurrently.

In a simple Java application, each client thread explicitly requests a connection, then executes statements on this connection. A more sophisticated application might use connection pooling, where a server component might request a single connection and use it to execute concurrent statements for multiple client threads. (The server component might also request a separate connection for each thread, although the initialization of each of these connections can consume quite a bit of overhead.)

By default, a connection automatically commits changes after executing each statement. However, you can disable auto-commit for a connection, and explicitly commit or roll back transactions using commit and rollback methods defined by the Connection class. All statements on the same connection reside in the same transaction space; they are all committed or rolled back together. Therefore, if statements for two logically separate transactions are executing concurrently on the same connection, the first transaction that commits or rolls back will commit or roll back all other current transactions.

To use multi-threaded database access safely, you must either open and close connections as they are needed by individual transactions and suffer the resultant performance degradation, or use a JDBC connection manager interface that manages a pool of connections for use by multiple transactions.

Transparent Persistence

To resolve some of the portability, synchronization, and concurrency limitations of the JDBC programming model, Forte for Java provides an alternative programming model, known as Transparent Persistence. Transparent Persistence, in addition to resolving JDBC limitations, also automates and manages persistence operations, making them generally easier to code than by using JDBC.

- Automation

  Transparent Persistence automates the mapping between persistent data objects and information in a data store, and also automatically generates database query and update code. The Transparent Persistence tools used for this automation accommodate a range of data stores, making persistence logic within an application not only transparent to programmers, but portable across various database systems.

- Persistence Management

  Transparent Persistence also provides runtime classes for managing persistence operations. The Transparent Persistence runtime classes not only perform persistence operations transparently (you do not have to write mapping code or write database-specific query and update statements), they also provide services for managing transactions, concurrency, and connection pooling.

The following sections provide a high-level introduction to the Transparent Persistence programming model. A full description of the Forte for Java Transparent Persistence features and programming model is provided in Chapters 3, 4, 5, and 6.

Transparent Persistence Programming Model

The Transparent Persistence programming model, unlike JDBC, automates most of the software layers identified in FIGURE 1-1.

- You don't have to explicitly obtain a connection to a data store.
- You don't need to write or execute SQL statements.
- You don't have to write mapping code.

Instead, the Forte for Java Transparent Persistence feature lets you view and manipulate persistent data stored in JDBC-compliant databases as Java objects, without the need to know SQL, the JDBC API, or database programming. You use

Transparent Persistence tools to create *persistence-capable* classes. These are classes used to represent persistent data and for which the Transparent Persistence runtime system can automatically perform and manage persistence operations.

To create persistence-capable classes, you use Forte for Java Transparent Persistence tools that generate class definitions from database schema or that map existing classes to database schema. The Transparent Persistence tools also *enhance* these classes so that the Transparent Persistence runtime can dynamically generate statements specific to the data store. These statements are used to perform persistence operations on the database to which the persistence-capable class was mapped.

FIGURE 1-3 illustrates the runtime objects involved in Transparent Persistence persistence operations. These objects are instances of classes that implement interfaces in the Transparent Persistence API. These objects are referenced by code in a persistence-aware component, also shown in FIGURE 1-3, that interacts with the Transparent Persistence runtime to perform persistence operations.

For example, to read data into a persistence-capable class instance, you obtain a Persistence Manager from a Persistence Manager Factory object, then obtain a Query from the Persistence Manager, pass it parameters, and execute it. In this case, the Transparent Persistence runtime system creates a collection of instances of the persistence-capable class and populates it with the results of the query.

Similarly, you can write values from a new persistence-capable class instance into the database by calling the `makePersistent` method of the Persistence Manager. The required connection, managed by the Connection Factory and the data store, generates the appropriate data-store-specific statements (based on the persistence-capable class definition) and sends them to the data store for execution.

You must perform any writing of data to the database in a transactional context. You do this by obtaining a Transaction object from the Persistence Manager. You use this object to begin a transaction, then commit or roll back the transaction. Any data manipulation of persistent instances between begin and commit is part of the same transaction. The transaction is entirely within your control.

Each Persistence Manager can support only one transaction. Thus, each thread that will perform a transaction generally obtains its own Persistence Manager. The Transparent Persistence runtime system, however, supports both concurrency management and connection pooling, allowing this system to scale appropriately.

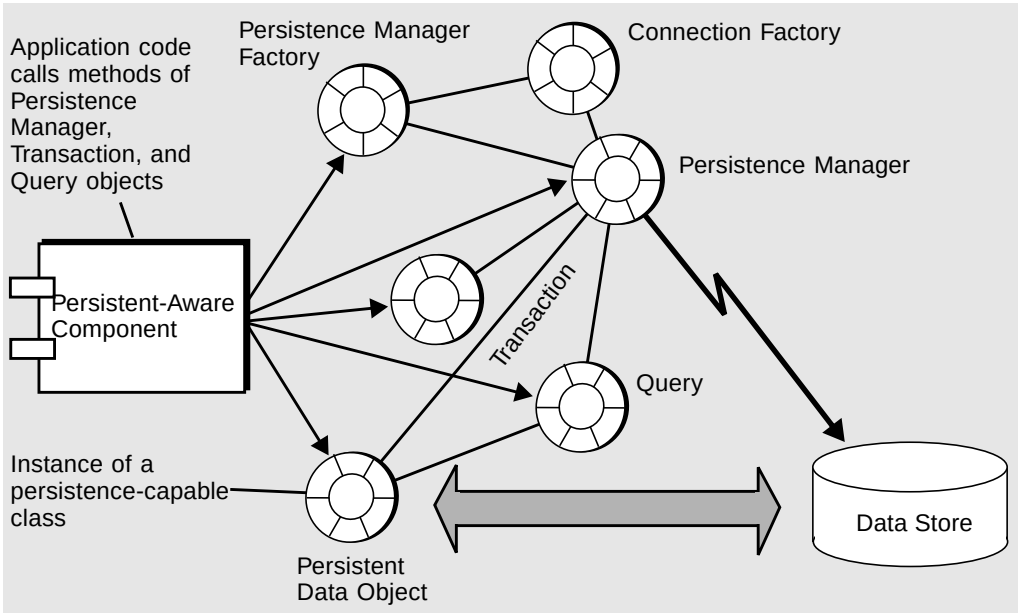

FIGURE 1-3 Transparent Persistence Programming Model

In the Transparent Persistence programming model, concurrency and connection management are performed by the Persistence Manager Factory and corresponding Connection Factory. You configure the Persistence Manager Factory for a particular data store and login name, and you can set properties such as the type of concurrency and connection management to be supported by the Transparent Persistence runtime system for each Persistence Manager instance.

- Concurrency

  You can choose between data store and optimistic concurrency. Data store concurrency uses the underlying database locking mechanism (if any) for the duration of the transaction, while optimistic concurrency allows for database reads to take place by multiple threads, but checks that no change has taken place to a database row before writing to it. Optimistic concurrency generally provides higher performance when multiple users are accessing the same data, and the duration between reading and updating the data is dependent on user "think time."

- Connection Management

  The Persistence Manager Factory can be configured to manage a connection pool, in which connections are shared and recycled among a number of Persistence Manager instances, thus optimizing on connection resources. Connection pooling provides for higher performance when large numbers of threads are accessing the same databases.

CHAPTER 2

Using Java Data Base Connectivity

Forte for Java provides a JDBC (Java Database Connectivity) module that automates many programming tasks that you use when building client components or applications that interact with a database.

The goal of the Forte for Java JDBC module is to increase your productivity when programming visual forms that contain Swing (Java Foundation Class) components that use JDBC to retrieve and update database tables. You can use this module to assist you in generating simple, two-tiered application architectures.

This chapter describes the following JDBC productivity enhancement tools provided by Forte for Java, and begins with a brief description of the steps you follow in creating a JDBC application. The tools include:

- Database Explorer
- JDBC JavaBeans components
- JDBC Form Wizard

Programming JDBC

This section provides a brief introduction to JDBC programming tasks, supplementing information provided in "JDBC Programming Model" on page 10.

General Programming Steps

When you perform JDBC programming, you follow these general programming steps:

1. Import relevant classes within your code.
2. Load a JDBC driver.

3. Establish a connection with a database.
4. Create a Main method.
5. Create try and catch blocks and retrieve exceptions and warnings.
6. Set up and use database tables.
    a. Create a table.
    b. Create JDBC statements.
    c. Execute Statements to perform persistence operations.
        i. Enter data into a table.
        ii. Obtain data from a table.
        iii. Create an updatable result set (`RowSet`).
        iv. Insert and delete rows programmatically.
    d. View changes in a `ResultSet` by managing the Transaction Isolation Level.

Forte for Java simplifies most of these tasks, generating JDBC code either through your editing of the Forte for Java JDBC JavaBeans component properties or through your use of the JDBC Form Wizard.

JDBC Reference Materials

While this chapter provides a discussion of JDBC programming in the context of the Forte for Java IDE, it assumes familiarity with the basics of the JDBC programming model. For additional information about JDBC, you can review the following reference materials, grouped by function.

Learning JDBC Programming

The Java Developer Connection provides an excellent tutorial on JDBC:

`http://developer.java.sun.com/developer/onlineTraining/new2java/programming/learn/jdbc.html`

In addition, the Java Developer Connection supplies a JDBC Short Course:

`http://developer.java.sun.com/developer/onlineTraining/Database/JDBCShortCourse/index.html`

Technical Articles

Sun has produced a document entitled:

"Duke's Bakery – A JDBC Order Entry Prototype – Part I":

```
http://developer.java.sun.com/developer/technicalArticles/
Database/dukesbakery/
```

Getting Started With JDBC

The following index is a reference when starting to program using JDBC:

```
http://developer.java.sun.com/developer/technicalArticles/
Interviews/StartJDBC/index.html
```

Another document is "Of Java, Databases, and Really Cool Dead Guys":

```
http://developer.java.sun.com/developer/technicalArticles/
Interviews/Databases/index.html
```

JDBC Basics

You can find additional information on JDBC within the Sun tutorial:

```
http://java.sun.com/docs/books/tutorial/index.html
```

This tutorial also provides some references:

```
http://java.sun.com/docs/books/tutorial/jdbc/basics/index.html
```

Using the Database Explorer

Before you begin the process of writing JDBC code, you need to understand the database that your application will use. To obtain database information, you can use the Forte for Java Database Explorer.

Using the Forte for Java Database Explorer, you can perform the following tasks:

- Browse database structures
- Examine all tables present in the database, including column and index information
- Examine SQL views related to the database

- Examine all stored procedures defined in the database
- View database data
- Create tables
- Create views
- Take “snapshots” of database structures
- Monitor SQL commands sent to the database
- Connect to a database

To learn how to perform these tasks, refer to the Database Explorer Help within the Forte for Java IDE.

Using JDBC Components

Forte for Java provides database connectivity and JDBC code generation tools for visual forms and components, specifically providing two basic types of components that you can use with your JDBC application:

- Visual Components—Swing components let you display tabular database information. Within Forte for Java, use Swing visual components to create forms that relay database data to the user; swing components provide the means to let you manipulate row data and display columns. Forte for Java generates the appropriate Swing code for you. Another type of visual component is a Data Navigator–a JDBC component that you add to a form to manipulate the display of data to the user.
- Non-visual components—JavaBeans components that do not have visual representation, but can be used to manipulate data from a database. One type of non-visual component is a `RowSet`, which is a type of row group that contains information from the database. To understand how to use JDBC JavaBean components, you need to:
- Understand the JDBC tab
- Understand how to program applications with JDBC components by:
  - Creating a Visual Form with Forte for Java
  - Using the Forte for Java Component Inspector with JDBC JavaBeans components

The JDBC Tab

The JDBC tab in the component palette contains icons for a number of JDBC JavaBeans components that you can use to facilitate the interaction of Java Swing components with a database. These components have properties that you customize using the Forte for Java Component Inspector.

The components include:

- `Connection Source`
- `Pooled Connection Source`
- `NB Cached RowSet`
- `NB JDBC RowSet`
- `NB Web RowSet`
- `Stored Procedure`
- `Data Navigator`

Connection Source

A Connection source is a non-visual component that provides a connection to a JDBC compliant database. When you configure the `Connection Source`, you set:

- database URL
- JDBC driver name
- user name
- password

Pooled Connection Source

A `Pooled Connection Source` component is similar to a `Connection Source`. However, when you specify the use of a `Pooled Connection Source` with your application, database connections that are established during application runtime are not closed when the application ceases to use the connection.

Instead, Forte for Java retains the connection in a pool for subsequent use within the runtime application. You can use a `Pooled Connection Source` when your application performs frequent open and close requests against a database to which it is connected.

Understanding RowSets

A RowSet component represents rows fetched from the database. You can use these components to configure data models for several Swing components.

`RowSet` *Background*

A `RowSet` object contains a set of rows from a JDBC result set or another source of tabular data, such as a file or spreadsheet.

Depending on how you implement them in your code, `RowSet`s can be serializable or extensible to non-tabular sources of data.

Because a `RowSet` object follows the JavaBeans model for properties and event notification, it is a JavaBeans component that can be combined with other components in an application.

`RowSet`s can be either connected or disconnected, depending on their implementation. A disconnected RowSet obtains a connection to a data source to fill itself with data or to propagate changes in data back to the data source, but most of the time it does not have a connection open.

Even when it is disconnected, a `RowSet` does not require the use of a JDBC driver or the full JDBC API, so its size is small. A disconnected `RowSet` is an ideal format for sending data over a network to a thin client.

Types of RowSets:

The JDBC Tab makes three different types of row sets available:

- `NB Cached RowSet`

  The `NBCachedRowSet` is a disconnected `RowSet` that caches its data in memory. This special type of `RowSet` is suitable for smaller sets of data. You can use it to create JDBC applications that provide code to operate on thin Java clients, such as Personal Digital Assistants (or PDAs).

  When a `RowSet` is disconnected from its data source, any updates that application writes on the `RowSet` are propagated to the underlying database.

- `NB JDBC RowSet`

  The `NBJDBCRowSet` represents a JavaBeans™ wrapping of a connected ResultSet object to be used in models of Swing components. It can be used to read extremely long tables more efficiently than a cached RowSet, which stores all data in an internal cache.

- `NB Web RowSet`

  The `NBWebRowSet` represents a set of fetched rows in a cache to be used in models of Swing components. It provides all cached RowSet functionality, and enables the rows to be imported and exported in XML format. The file can then be sent over the internet using HTTP/XML protocols.

You can customize a JDBC `RowSet` by setting the following properties under the properties tab in the Properties Editor:

TABLE 2-1 `RowSet` Properties

Property	Definition
Command	SQL query to populate this `RowSet`. The query can be any syntactically-correct SQL Select Query.
Connection provider	The configured connection source; a drop-down list provides choices.
Read-only	If True, this `RowSet` is read-only. Data from the `RowSet` cannot be written out to the database.
Rowcount	The number of rows.
Status	Status of a read against a `RowSet`
Transaction isolation	determines how the `RowSet` handles data under transactions. For detail, see Java documentation for `java.sql.Connection`.
XML output directory (`WebRowSet` only)	Identifies the directory where data from the `WebRowSet` will be sent.
XML Output File (`WebRowSet` only)	Determines the name of the file that will contain the XML output from a `WebRowSet`.

Other Properties, Event, and Code Generation Tabs for a `RowSet`

The Other Properties Tab for a `RowSet` enables you to inspect and modify additional properties.

TABLE 2-2 `RowSet` Other Properties Tab Properties

Property	Definition
Database URL	The location of the database where records will be updated. In most cases, it is the same URL as listed in the Database URL property of Connection Source.
Default Column Values	The values to be inserted into a new row. You can press Fetch Columns to retrieve a list of columns in the RowSet.
Execute on load	If `true`, the `NB RowSet` can be executed on load. You can specify a parameter with the Execute on Load from a Form Connection, and you can generate initialization code.

TABLE 2-2 `RowSet` Other Properties Tab Properties *(Continued)*

Property	Definition
Password	A password the user must supply to gain access to the table that contains this `NB RowSet`.
Table Name	The name of a database table where records will be updated.
User Name	The name of a user updating records.

The Event Tab for a RowSet enables you to inspect and modify events associated with `RowSet`s.

TABLE 2-3 `RowSet` Event Tab Properties

Property	Definition
`cursorMoved`	Specifies event handlers for the `cursorMoved` event. This method is called when an `NBCachedRowSet`'s cursor is moved.
`rowChanged`	Specifies event handlers for the `rowChanged` event. This method is called when a row in a `RowSet` is changed.
`rowInserted`	Specifies event handlers for the `rowInserted` event. This method is called when a row in a `RowSet` is inserted.
`rowSetChanged`	Specifies event handlers for the `rowSetChanged` event. This method is called when an `RowSet` is changed.
`rowCompleted`	Specifies event handlers for the `rowCompleted` event. This method is called after an inserted row is committed to the database.

The Code Generation Tab enables you to specify pre- and post-processing code related to a rowset.

TABLE 2-4 Code Generation Tab Properties

Property	Definition
Code Generation	Choose between generating standard or serialization code for the component.
Custom Creation Code	Enter your own creation code for the component, not including the variable name and equal sign (=). This creation code is called in the `initComponents()` method. If this property is left blank, the IDE generates a default creation code for the component.
Post-Creation Code, Post-Init Code, Pre-Creation Code, and Pre-Init Code	Write custom code that you want the IDE to place before and after a component's creation code and before and after its initialization code. The IDE always places creation code before initialization code in `initComponents()`.

TABLE 2-4 Code Generation Tab Properties *(Continued)*

Property	Definition
Serialize To	Set the name of the file for the component to be serialized to, if it is serialized.
Use Default Modifiers	Set to `True` if you want the component's variable modifiers (public, private, and so on) to be generated using the default modifiers. The default modifiers are specified in the Variables Modifier property of the Form Objects node in the Options window. (Choose Tools > Options to view the window.) Set to `False` if you want the Variables Modifier property to appear on the component's property sheet, enabling you to override the default modifiers.
Variable Name	Modify the component's variable name.

Data Navigator

The JDBC module provides a visual component that provides direct navigation of a `RowSet` with a pre-built GUI. This component is useful when you need to create prototypical applications and when you want to create data entry applications.

You can customize a Data Navigator by setting the following properties under the properties tab in the Properties Editor of a Data Navigator.

TABLE 2-5 Data Navigator Properties

Property	Definition
AutoAccept	Automatically accept changes in the database. When you specify this property, changes you make through the Navigator are either immediately propagated to the database, or added to the RowSet and propagated to the database when you request it.
Bound RowSet	The `RowSet` to be controlled by the Data Navigator.
Layout of buttons	Determines whether buttons are displayed in one or two rows.
Modification buttons	Enables or disables the display of buttons for modification.

Stored Procedures

Stored procedures are a group of SQL statements that form a logical unit and perform a specific task. Stored procedures encapsulate operations or queries that execute on a database server. Such procedures, of course, vary in their nature according to the database management system (DBMS) on whose server they execute.

Within the Forte for Java IDE, a stored procedure is a non-visual component that represents a database stored procedure in your JDBC application. You can call a stored procedure in response to an event initiated by a user within an application GUI (such as a button click).

The syntax for a stored procedure is different for each database management system that Forte for Java supports. For example, one database management system might use `begin`, `end`, or additional keywords to indicate the beginning and ending of the procedure definition, while a second DBMS might use other keywords to indicate the same parts of the procedure definition.

The *JDBC Tutorial* provides information on some of the stored procedures you can create for different databases, in addition to information on calling a stored procedure from your JDBC application.

You can customize a stored procedure by setting the following properties under the properties tab in the Properties Editor of a stored procedure. Once you have specified these properties in the property sheet, you can connect stored procedures to any user action.

TABLE 2-6 Stored Procedure Properties

Property	Definition
Arguments	Represents database data that you want used by the stored procedure when called from the application.
Bound RowSet	Enables you to select a RowSet from a drop-down list that is refreshed from the database after the stored procedure is called.
Call format	Format in which your stored procedure is called. For example, it might include Name and Arguments that are substitution codes for the properties with those names on this property sheet.
Connection provider	A configured connection source in whose context the stored procedure is to be called from the application.
Name	The name of your called stored procedure.

Programming With JDBC Components

Use the visual and non-visual components provided in the JDBC module in conjunction with Swing components to create forms that you use to retrieve and manipulate database data.

For example, a number of Swing components (`JList`, `JTable`, `JComboBox`, `JButton`, `JToggleButton`, `JRadioButton`, `and JCheckbox`) are associated with data models for the data they display. Within the IDE, you use Property Editors and the Component Inspector to customize the data model for these Swing components

by specifying the JDBC components with which they interact to access a database. After you have completed specifying the JDBC components, Forte for Java generates the corresponding JDBC code.

Setting Data Models for Components

The following Swing components have associated data models.:

- `JList`
- `JTable`
- `JComboBox`
- `JButton`
- `JToggleButton`
- `JRadioButton`
- `JCheckbox`

You can configure these data models to use data from the database.

The most common component to display database tables is `JTable`. The model can be configured in the property sheet of each Swing component (under the model property).

Selecting Database Columns

Components that can display multiple rows, such as `JTable` or `JList`, also have the `selectionModel` property.

`JList` and `JComboBox` also have a special kind of model. This model consists of using one column from one RowSet to work with another column from another RowSet to display data, using a SQL join. See below for details.

Text components which have the document property (such as `JTextField`, `JTextArea`, `JPasswordField`, `JTextPane`, and `JEditorPane`) can set up this property to use data from the database.

▼ To Configure the Data Model for `JTable`

1. **For the model property in the `JTable`'s property sheet, open the custom property editor by clicking on the value of the property and then clicking the ellipsis (...) button that appears.**
2. **Choose the TableEditor mode.**
3. **In the RowSet field, choose the RowSet to be displayed in the table.**
4. **Use Fetch columns to load column names into the list.**

5. **Use the Add, Remove, Edit, Move Up, and Move Down buttons to set the names and order of the columns in the table.**

6. **Click OK to preserve the changes and close the custom property editor.**

▼ To Configure the Selection Model for `JTable` and `JList`

1. **For the selectionModel property in the component's property sheet, open the custom property editor by clicking on the value of the property and then clicking the ellipsis button (…) that appears.**

2. **In the RowSet field, choose the RowSet to be displayed in the table or list.**

3. **Click OK to preserve the changes and close the custom property editor.**

▼ To Configure the Data Model for `JList` and `JComboBox`

1. **For the model property in the component's property sheet, open the custom property editor (by clicking on the value of the property and then clicking the ellipsis button (…) that appears).**

2. **For the Primary RowSet fields, choose the RowSet for the data model to retrieve rows from, and then select one column from the Column drop-down list.**

3. **If you want, in the Secondary RowSet field, choose the RowSet to display data from (according to a SQL join). Corresponding columns from the primary and secondary RowSet must have the same data type.**

4. **If the Join check box is checked, a corresponding component displays the result of a database join. If it is unchecked, a corresponding component is used as a code map to set values in the primary rowset.**

5. **Choose a Data column (join column) and Display column (visible data). Click OK to preserve the changes and close the custom property editor.**

▼ To Configure the Data Model for `JCheckbox`, `JRadioButton`, and `JToggleButton`

1. **For the model property in the component's property sheet, open the custom property editor (by clicking on the value of the property and then clicking the ellipsis (…) button that appears).**

2. **Choose the RowSet from which the data is to be fetched.**

3. **Choose a column; data from this column will be used to decide if the component should be selected.**

4. **Enter the database value corresponding to a selected component into the Select field and the value of an unselected component into the Unselect field.**

5. **Click OK to preserve the changes and close the custom property editor.**

▼ To Configure the Document Model for Text Components

1. **For the document property in the component's property sheet, open the custom property editor by clicking on the value of the property and then clicking the ellipsis button (…) that appears.**
2. **Choose the RowSet from which the data is to be fetched.**
3. **Choose a column in which to display the text component.**
4. **Click OK to preserve the changes and close the custom property editor.**

Creating a Visual Form

After you have used the Property Editor to customize Swing components in your application, Forte for Java enables you to create a visual form associated with the Swing components that interacts with the database.

▼ To Create a Visual Form With Swing Components That Interact With a Database

1. **Create a Swing component form using a template provided in the Forte for Java IDE.**
2. **Add any needed** `Connection Source` **(or** `Pooled Connection Source`**),** `RowSet`**, or** `Stored Procedure` **nonvisual components to your form from the Component Palettes.**
3. **Using the corresponding Property Editor, customize these components for the database entities they represent.**
4. **Add any visual components you need, including the Data Navigator.**
5. **Use the corresponding Property Editor to customize the visual components appropriately, referencing the** `RowSet` **components you need.**

   As you specify the Swing components to use with your JDBC application, Forte for Java automatically creates the correct Swing classes to use in your application.
6. **Use the Properties Editor for the specified form to indicate exceptions that should be caught during runtime and run the form.**

Using the Component Inspector With JDBC Components

You can use the Forte for Java Component Inspector to modify properties for components you use in your JDBC application. The following components can be found under Non-visual Components in the Component Inspector:

- `NB Cached RowSet`
- `NB JDBC RowSet`
- `NB Web RowSet`
- `Connection Source`
- `Pooled Connection Source`
- `Stored Procedure`

The `Data Navigator` component and other Swing components are shown according to their position in the container hierarchy.

Using the JDBC Form Wizard

The JDBC Form Wizard guides you through the creation of a form that can interact with database tables. It provides a substitute for the explicit editing of properties that you would otherwise perform if you used the approach outlined in "Using JDBC Components" on page 18. When you finish running the wizard, you will have a generated application, a file name for the application, and a package.

The following sections illustrate the JDBC Form Wizard, using the sample PointBase Server Database that comes included with the Forte for Java IDE.

▼ To Open the JDBC Wizard

- **Select Tools > JDBC Form Wizard**

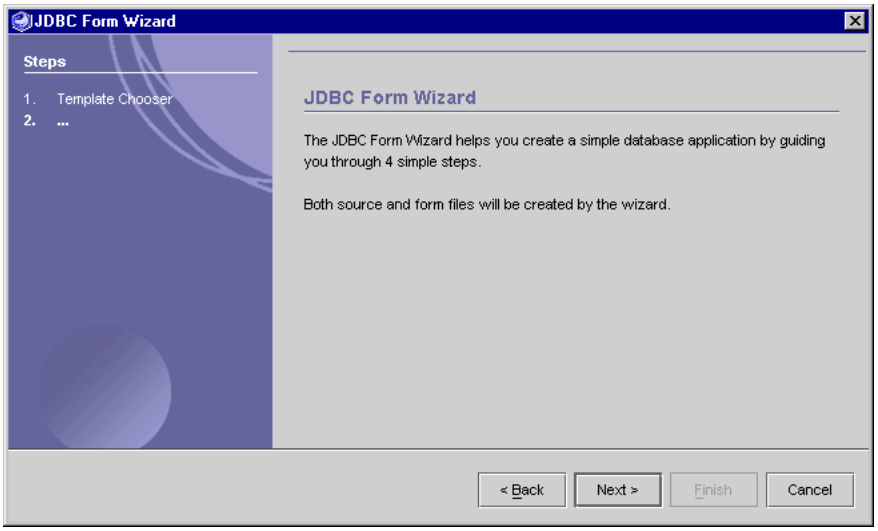

FIGURE 2-1 JDBC Form Wizard, Opening

Establishing a Connection

When you use the JDBC Form Wizard or when you use the JDBC tab to create a JDBC client application, one of the first tasks you must perform is to establish a connection with the database management system that you want to use.

Typically, the JDBC Form Wizard or Forte for Java connection generates the code that you can use in your JDBC application when you use the Visual Form Editor or the JDBC Form Wizard to create a form. The application uses the form to populate information that it obtains from a database management system.

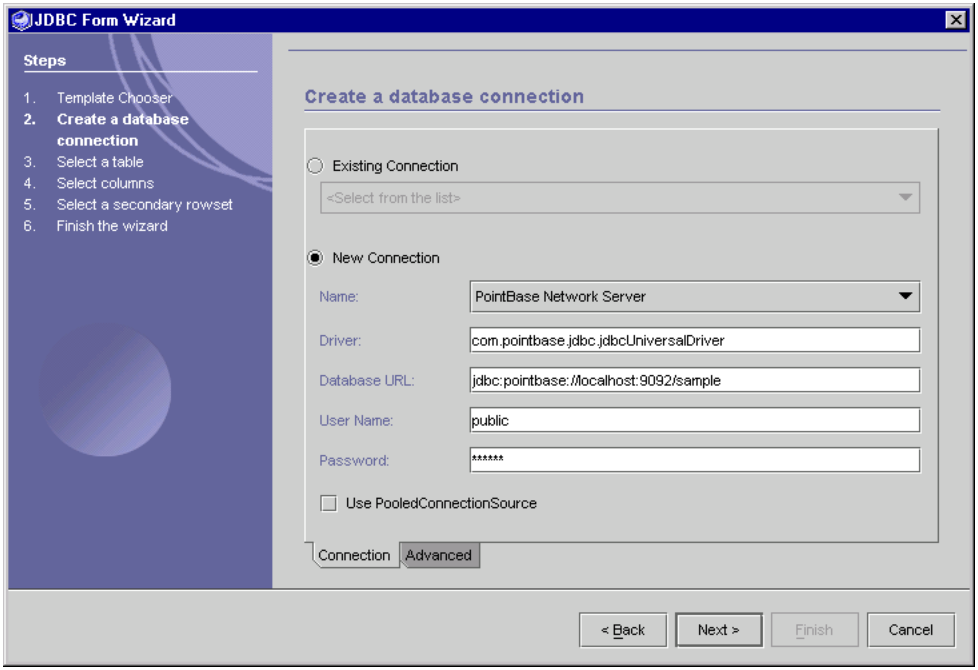

FIGURE 2-2 JDBC Form Wizard, Database Connection

The second panel of the JDBC Form Wizard lets you establish a connection with a database. You can specify the use of a pooled connection for a DataSource in this panel.

When you need a new connection, you must supply:

- The name of your database. For example, PointBase Network Server.
- The JDBC driver name for the database. For example, `com.pointbase.jdbc.jdbcUniversalDriver`.
- The Database URL where the database is located. For example, `jdbc.pointbase://localhost:9092/sample`.
- User Name
- Password
- Select the Use Pooled Connection Source check box to specify an optional pooled connection.
- Optionally select the Advanced tab to specify a schema to get tables.

Forte for Java provides these parameters to the JDBC application code that it generates.

You can select an existing connection by clicking the Use Existing Connection radio button, and selecting the connection from the drop-down list.

When you select the Next button, Forte for Java calls a method that creates a database connection based on parameters you enter. You use this connection to the database in the same way that you use the wizard to write JDBC application code.

Selecting Database Tables or Views

The third panel of the JDBC Form Wizard lets you:

- Select a table or view in the database to which you are connected.
- Specify that you want only read access to a specific table for your generated JDBC application. This means that the application cannot alter data in the database.
- Add a `rowInserted` event handler to a table. This event handler handles the listening for events associated with the application's insertion of rows into the tables you select.
- Set the Transaction Isolation level for a table. See "Transaction Isolation Levels" on page 32.
- Provide a SQL command to run against the tables you specify.

The JDBC Form Wizard lets you execute SQL statements against tables you specify in the Wizard. You use the data from the SQL output to populate visual forms. You can specify SQL statements which, when applied to a specific form, generate the appropriate SQL code. In FIGURE 2-3, Forte for Java provides a default SQL command to use with the table you have selected.

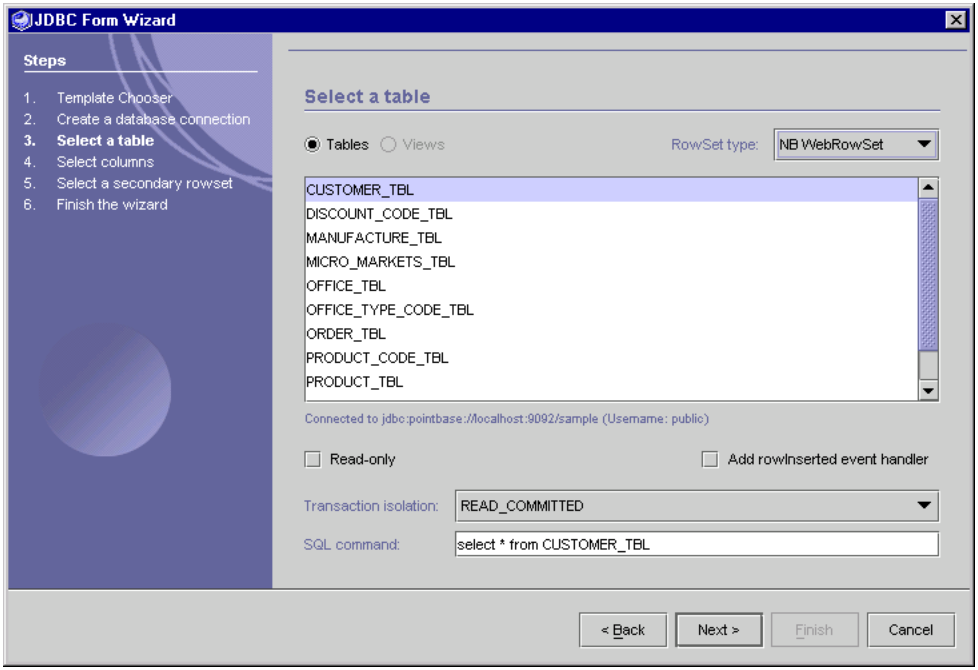

FIGURE 2-3 JDBC Form Wizard, Select a Table

Transaction Isolation Levels

To avoid conflicts during a transaction, a database management system uses *locks*. Locks are operative until the application commits the transaction or rolls it back from the database.

Locks are set according to a transaction isolation level. Locks apply to the entire `ResultSet` that is returned to the application or committed from the application to the database.

Each database management system provides its own default transaction isolation level. Forte for Java lets you choose between the transaction isolation levels within the second panel of the JDBC Form Wizard.

Note – The driver and the data base management system must support the transaction isolation level you use.

TABLE 2-7 Transaction Isolation Levels

Property	Definition
`TRANSACTION_READ_COMMITTED`	Prohibits a transaction from reading a row that has uncommitted changes in it.
`SERIALIZABLE`	Includes the prohibitions in `TRANSACTION_REPEATABLE_READ`. It prohibits the situation where one transaction reads all rows that satisfy a `WHERE` condition, a second transaction inserts a row that satisfies that `WHERE` condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.
`TRANSACTION_NONE`	Transactions are not supported.
`TRANSACTION_REPEATABLE_READ`	Prohibits a transaction from reading a row with uncommitted changes in it. It also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time (that is, a non-repeatable read).
`TRANSACTION_READ_UNCOMMITTED`	A row changed by one transaction can be read by another transaction before changes in that row are committed to the database. If changes are subsequently rolled back, the second transaction retrieves an invalid row.

Selecting Columns to Display

The fourth panel of the JDBC Form Wizard lets you select columns from the database tables to include in the form that is displayed. In this panel, you can specify:

- Columns you want displayed in the application you generate
- The order of the columns you want displayed
- Column parameters:
  - Column title
  - Column editability
  - Default column value
  - A Swing component to display the table in the application

In the example provided, `JTable` (the most common Swing form) is used. The `JTable` form displays more than one column of data in the application.

Other Swing component choices include:

- `Jlist`: displays a column in a list
- `JComboBox`: displays one column in a combo box
- `JTextField`: displays one or more columns in a text field

In FIGURE 2-4, the first Column is selected. It can be removed or moved in position.

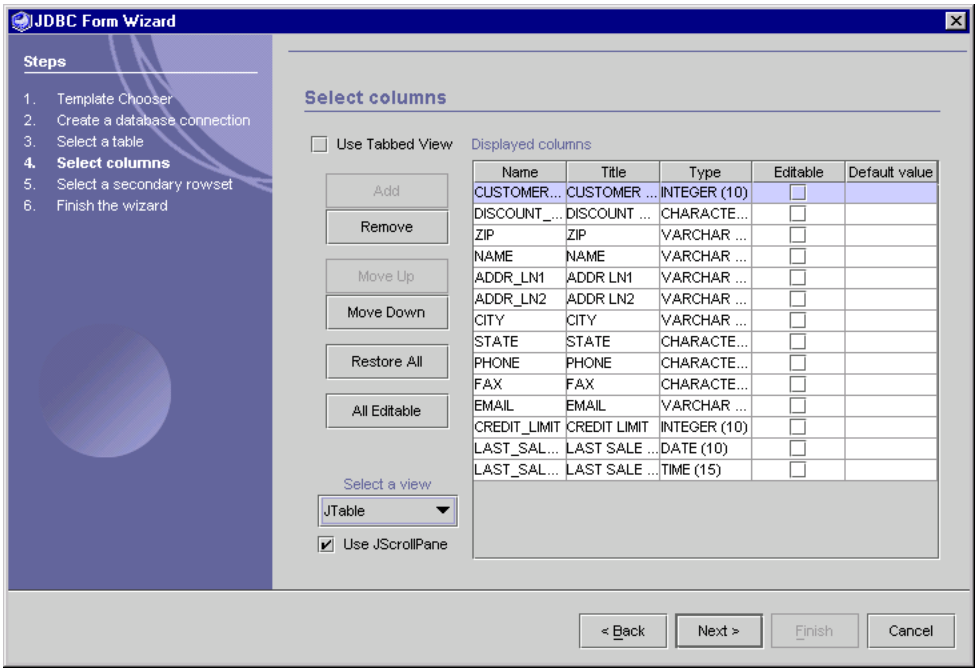

FIGURE 2-4 JDBC Form Wizard, Select Columns

If you choose `JList` or `JComboBox`, only one column can be displayed, and you can choose a column to display from the Name property:

1. **Select a value in the Name column.**
2. **Select a column name from the built-in combo box.**

▼ To Edit Column Titles

1. **Click on the Title field you want to edit. An edit window appears with two tabs.**
2. **Select the String Value tab to enter the new name as a simple string value.**
3. **Select Resource Bundle to enter the name using a resource bundle. Enter the name of the bundle into Bundle Field, and select any related keys from the Keys combo box.**

4. **Select OK to close the edit window.**

Selecting a Secondary `RowSet`

This panel displays a list of all available tables according to the database connection created on the Connection panel and is enabled only if a view supporting two `RowSet`s (`JList` of `JCheckbox`) is selected.

You can use this panel to populate the secondary `RowSet` of the generated application.

▼ To Select a Secondary `RowSet`

1. **Check Use Secondary Rowset.**

   If you check this rowset, the secondary rowset is used in the generated application.

2. **Select either the Tables or Views radio button.**

3. **Select a type of rowset from the RowSet type combo box.**

4. **Select a table or view from the list.**

5. **Check Read-only if you want the corresponding rowset to be read-only.**

6. **Check Add rowInserted event handler to add a rowInserted event handler to the source code of the generated application.**

   The handler is called when a new row is inserted and enables the creation of default column values dynamically.

7. **Choose a transaction isolation level for the rowset using one of the values in the Transaction isolation combo box.**

   The default transaction level is `READ_COMMITTED`.

8. **Use the SQL_command text field to prepare SQL to populate the rowset.**

   By default, Forte for Java generates the text `select * from table-name`.

9. **Select a data column to use with a database join.**

   Selecting this column will display a different field other than the primary column retrieved; however, it must be of the same data type as the primary column.

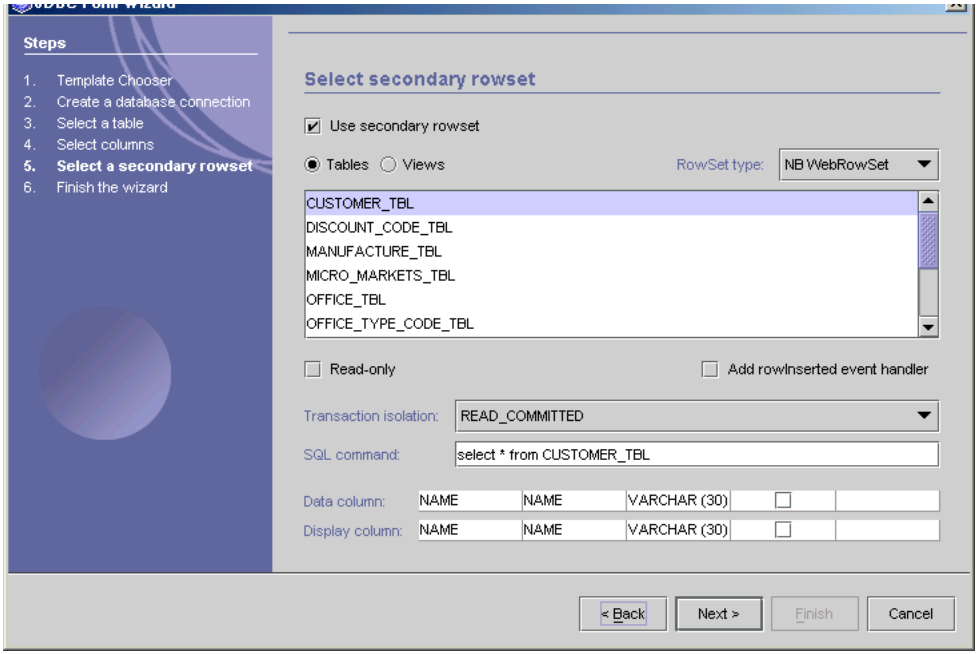

FIGURE 2-5 JDBC Form Wizard, Select Secondary RowSet

Previewing and Generating an Application

The last panel shows a preview of a generated application. Use this panel to complete your generated application. In addition, you can select a package and a file name to create a completed application.

Provide the name of the package under Package and the target file under Target.

You can view the component layout and the layout from the view of the Data Navigator. What you view depends on the Swing form you have chosen to contain the data that is manipulated in your application.

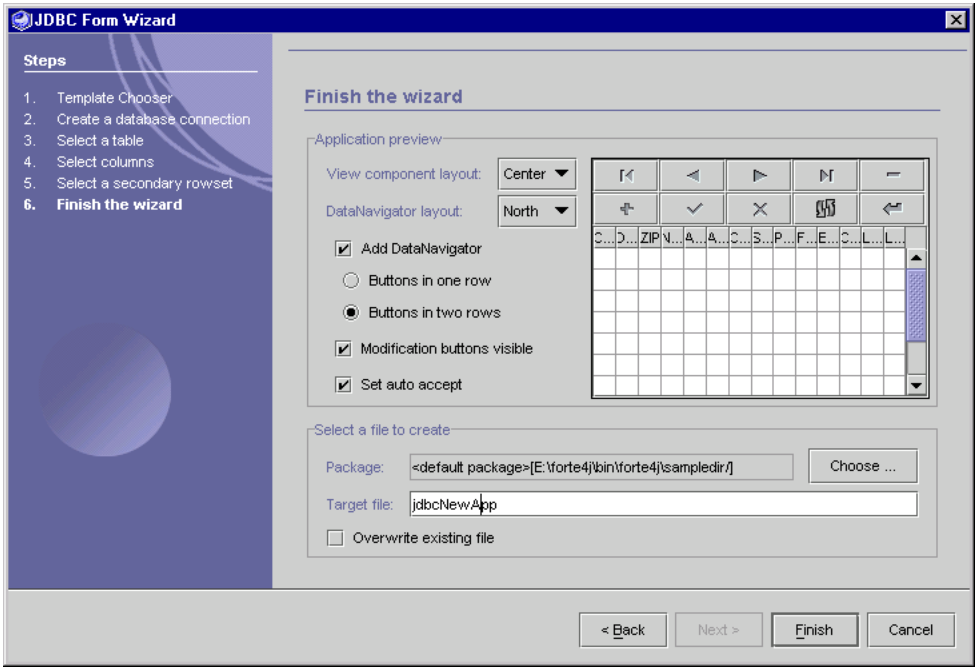

FIGURE 2-6 JDBC Form Wizard, Finish the Wizard

Running Your JDBC Application

You can compile, run, and debug JDBC applications as if they were any other form. If you need special JDBC drivers, ensure they are in Forte for Java's CLASSPATH, so they will, by default, be available for external compiling, executing, and debugging of JDBC-based forms.

You can run your application external to the IDE by adding paths to these packages into your CLASSPATH:

- `modules/ext/sql.jar`
- `modules/ext/rowset.jar`
- `lib/ext/jdbc20x.zip`
- A corresponding JDBC driver. JDBC drivers are typically stored in lib/ext.

If a `WebRowSet` is used in your JDBC application, two more JAR files are required:

- `lib/ext/parser.jar`
- `lib/ext/xerces.jar`

CHAPTER 3

Transparent Persistence Overview

The Forte for Java Transparent Persistence feature lets you view and manipulate persistent data stored in JDBC-compliant databases as Java objects, without the need to know SQL, the JDBC API, or database programming. This chapter provides a brief overview of the Transparent Persistence programming model.

Whenever you see the terms *classes*, *fields*, and *objects* in this manual, they refer to classes, fields, and objects for the Java platform.

What Is Transparent Persistence?

Transparent Persistence allows you to access information in data stores as Java objects, allowing for the separation of Java programming from database programming. This is done through persistence-capable Java classes, which contain data from a persistent data store, eliminating the need for SQL or coding specific to a particular data store.

Using Transparent Persistence and its mapping capabilities, you start with a relational database and map the columns of relational tables to automatically-generated or pre-existing Java classes. Transparent Persistence generates relationships between the Java classes that correspond to relationships between database tables. Tables and columns that are linked in the database by foreign keys are similarly connected in Java classes using reference or collection relationships.

Applications access the data store through operations on objects using the Java programming language, without knowing the database schema or using special database access languages. You can insert business logic into these Java programming language classes by defining additional methods and extending the automatically generated methods.

Transparent Persistence lets you map Java classes to a database schema automatically, using either of two methods:

- Database->Java mapping

  This method generates Java classes from a database schema, creating persistence-capable classes mapped to any or all tables in the schema. This approach is best if you do not yet have any classes to be mapped.

- Meet-in-the-middle mapping

  This method creates a custom mapping between an existing schema and existing Java classes. Use this approach if you already have classes that you want to use to access persistent data. You can also use it to fine-tune classes generated by Database->Java mapping.

Transparent Persistence also has a set of runtime libraries accessed by the Transparent Persistence API. This API is a set of Java classes for accessing the persistent objects from the underlying database, providing the framework for running the mapped Java classes.

Application developers can work with a set of Java classes that represent the persistent data their applications need. When an application needs to get data, the developer calls methods of a Persistence Manager or Query instance, which returns instances of persistence-capable classes. Another way to obtain data from the datastore is to navigate reference or collection relationships among persistent instances. When the application needs to change data it calls methods of the persistence-capable instances.

The Forte for Java Transparent Persistence module is a preview implementation of the forthcoming Java Data Objects (JDO) specification. A JDO implementation is a scalable, portable implementation of the Persistence Manager and other pieces of the JDO environment defined in the specification. Each JDO implementation enables persistence-capable classes to interact with some types of database software, connection managers, and so on.

Programming Transparent Persistence

Transparent Persistence anticipates two different types of developers, one with data store knowledge and the other with application knowledge, each working on different tasks:

- Developing persistence-capable classes
- Developing persistence-aware applications

Developing Persistence-Capable Classes

As a developer creating persistence-capable classes, you create a set of classes that model the data in a persistent data store. Chapter 4 describes the wizards you can use to develop these classes.

▼ To Create Java Packages From a Database Schema

1. **Capture a database schema using the schema capture tool.**

   This creates a file system representation of the database schema that you can use without a live connection to the database.

2. **Map persistence-capable Java classes to your database schema, using one of the following methods:**

   - Use the Java Generator wizard to generate new Java classes from the captured database schema tables along with a mapping from the generated classes to the schema's tables.
   - Use the Map to Database wizard to make existing Java classes persistence-capable, and map the database schema to those classes. You can also use this wizard to customize an existing mapping. For example, you could unmap a field, map a newly added field, map the class to a table in a different schema, or modify the mapping after changing and recapturing a schema.

3. **Add business logic to generated classes.**

   Edit the source code for the Java classes that correspond to database data. Typically, you add your business logic to these classes. You might add code to an existing or generated method, or you might add additional methods to these classes.

4. **Compile the source code files.**

   After coding is complete, compile the Java class source files using the Forte for Java IDE. These are the classes representing database tables.

5. **Archive or package the persistence-capable and persistence-aware classes.**

   Package the classes into the `.jar` file (either for deployment or another development stage that will not change persistence-capable classes) inside Forte for Java. Forte for Java will determine whether theses classes are persistence-capable or persistence-aware classes and enhance them for Transparent Persistence before adding them to the `.jar` file. The Enhancer automatically adds all the necessary support to the byte-code of the class to enable the class to cooperate with the Transparent Persistence runtime upon accessing persistent fields.

Note – If you choose to run or debug the application inside Forte for Java using the Persistence Executor or Persistence Debugger, the byte-code enhancement will be done by a special class loader; in this case, there's no need to package the persistence-capable or -aware classes in a `.jar` file.

Developing Persistence-Aware Applications

As a developer creating persistence-aware applications, you need to know which persistence-capable classes model the application domain data, and the standard Transparent Persistence API for working with those classes. These standard calls allow the you to select, update, insert, and delete data from the data store. These calls are discussed in Chapter 5.

After you have persistence-capable Java classes corresponding to database tables, you can write applications that use those Java classes. When you use the mapped classes, all of the necessary JDBC statements are generated for you automatically. You are responsible for transaction demarcation and specifying queries to find objects of interest in the database. The query is a Java expression-like boolean filter that is translated into an SQL select statement. See "Querying the Database" on page 113 for more information on writing queries.

Mapped Java classes can also be accessed directly in Java Server Pages (JSP™) using Transparent Persistence tags provided as part of JSP. These tags are discussed in Appendix B, and in *Building Web Components*.

Transparent Persistence and Enterprise JavaBeans

Enterprise Java Beans™ (EJB™) is a component architecture for development and deployment of distributed business applications. Transparent Persistence supports integration with Enterprise JavaBeans components in the following areas:

- With Stateful and Stateless session beans as the persistence-aware components that use persistence-capable classes directly as dependent objects;
- With Bean-Managed Persistence Entity Beans as persistent components that use persistence-capable instances as delegate objects to actually implement business methods by accessing and possibly modifying persistent state.

Container-Managed Persistence Entity Beans are not supported in this release.

The integration with Enterprise JavaBean components is described in more detail in Chapter 6.

CHAPTER 4

Developing Persistence-Capable Classes

This chapter describes how to use Transparent Persistence to map between a set of Java programming language classes and a relational database.

Mapping Capabilities

Mapping refers to the ability to tie an object-oriented model to a relational model of data—the schema of a relational database. Transparent Persistence provides the ability to tie a set of interrelated classes containing data and associated behaviors to the interrelated meta-data of the relational model. You can then use this object representation of the database to form the basis of a Java application. You can also customize this mapping to optimize these underlying classes for the particular needs of an application.

The result is a single data model through which you can access both persistent database information and regular transient program data. Application developers need only understand the Java programming language objects; they do not need to know or understand the underlying database schema.

The mapping changes you make here affect only the Java classes; the database schema remains as currently defined. The database schema and the Java classes are separate entities, as FIGURE 4-1 illustrates.

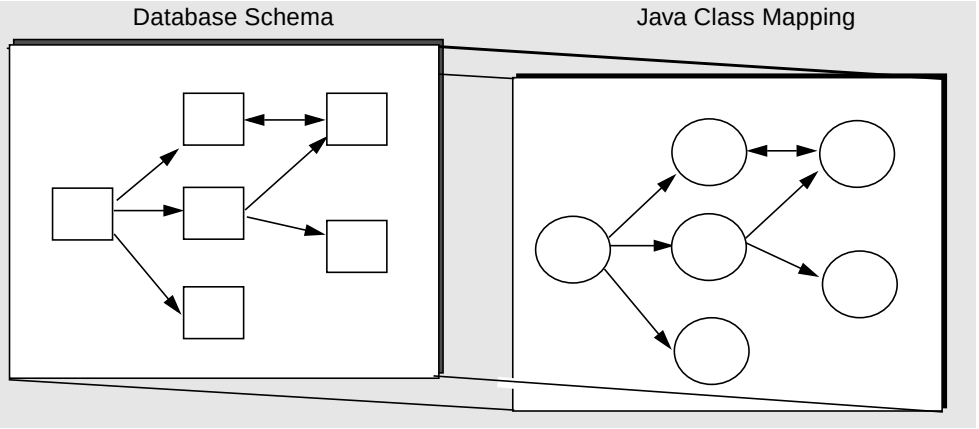

FIGURE 4-1 Mapping a Database to Java Classes

You can either generate both the mapping and the class model from the schema, or map an existing set of classes to an existing schema.

Note – Transparent Persistence maps each class to tables within a single database schema. All related classes must also map to that schema.

Mapping Techniques

A persistence-capable class should represent a data entity, such as an employee or a department. To model a specific data entity, you add persistent fields to the class that correspond to the columns in the data store.

The simplest kind of modeling is to have a persistence-capable class represent a single table in the data store, with a persistent field for each of the table's columns. An `Employee` class, for example, would have persistent fields for all of the columns found in the data store's `EMPLOYEE` table, such as `lastname`, `firstname`, `department`, and `salary`.

The class developer can also choose to have only a subset of the data store columns used as persistent fields.

You can use Transparent Persistence to map Java classes to a database schema using one of two techniques:

- Database to Java mapping

  This technique generates Java classes from a database schema, using the Generate Java wizard. The wizard creates persistence-capable classes mapped to any or all tables in the schema. This approach is best if you do not yet have any classes to be mapped.

  In this scenario, you need only to choose which of the tables in the schema will be mapped. During the modeling process, Transparent Persistence analyzes the schema, including primary key fields and the foreign keys fields that define relationships, and creates Java representations of them. The resulting set of objects reflects the organization of the meta-data in the database. The Java code is generated automatically.

- Meet-in-the-middle mapping

  This technique creates a custom mapping between an existing schema and existing Java classes, using the Database Mapping wizard and the Properties window. You should use this approach if you already have classes that you want to use to access persistent data. You can also use it to modify classes generated by the previous method.

Mapping Relationships

A relationship can be one-to-one, one-to-many, or many-to-many, depending on the number of instances of each class in the relationship. Relationships allow you to navigate from one object to its related objects. In the database, this might be represented by foreign key columns and, in the case of many-to-many relationships, join tables. In the Java code, relationships are represented by object reference—either collections or persistence-capable type fields, depending on the relationship cardinality.

When Transparent Persistence generates Java code, a collection field represents the many side of a one-to-many relationship. Transparent Persistence uses a variable of the actual persistence-capable class type to represent the single side of a one-to-many relationship.

For example, suppose you have a department object with a relationship to a collection of employees. You can navigate the relationship from the department object to see all the employees associated with that department. Similarly, you can view an employee and also see the department to which it is connected. Many employees can exist for a department, but there can be only one department per employee. The database uses a foreign key to make this connection.

Continuing the example, the `Department` class could contain an `employees` field of the type `HashSet`. This `HashSet` field gives the department object the ability to represent many employees. In addition, the `Employee` class contains a `department` field of the type `Department`. The `Department` reference field allows an employee to have one department.

The `Department` class would contain the following code:

```
private java.util.HashSet employees;
```

The `Employee` class would contain the following code:

```
private Department department;
```

Relationship fields appear under the Fields node for their class. The fields have some extra properties to indicate the related class, upper bound, lower bound, and so on. For meet-in-the-middle mapping, these properties are not set. You need to set them in the Properties window. See "Setting Options and Properties" on page 71 for more information.

You can either create a relationship automatically, through the Java Generation wizard, or by creating the correct type of field in the Java code.

Note – During Java generation, Transparent Persistence ignores a relationship field when that field references an unmapped class. In such a case, the Transparent Persistence module treats the relationship fields as ordinary fields.

The Java Generation wizard uses foreign keys from the database tables to determine relationships. It interprets a join table as a table with foreign keys that refer to different tables.

For example, suppose you have a `DEPARTMENT` table and an `EMPLOYEE` table with a one-to-many relationship between `DEPARTMENT` and `EMPLOYEE`. Both tables have primary keys. In addition, the `EMPLOYEE` table has a separate foreign key column that contains values corresponding to the `DEPARTMENT` primary key, `DEPID`. From this schema, Transparent Persistence generates a `Department` class and an `Employee` class. The `Department` class contains a field that can hold many employees, while the `Employee` class contains a field that can reference only one department. FIGURE 4-2 illustrates this.

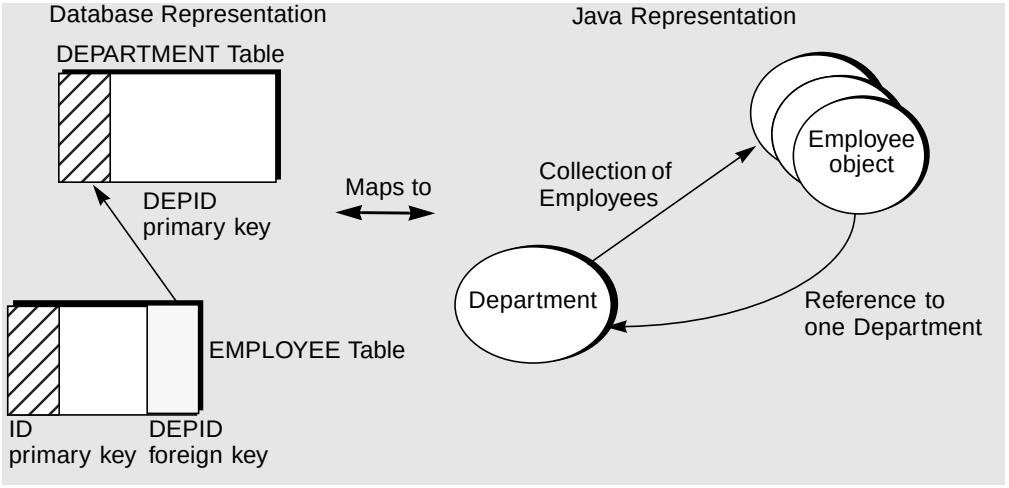

FIGURE 4-2 Foreign Keys and One-to-Many Relationships

The database uses join tables to represent tables in a many-to-many relationship. On the Java side, the classes at both ends of the relationship use fields that can hold multiple references to the other objects. FIGURE 4-3 shows how a many-to-many relationship might look.

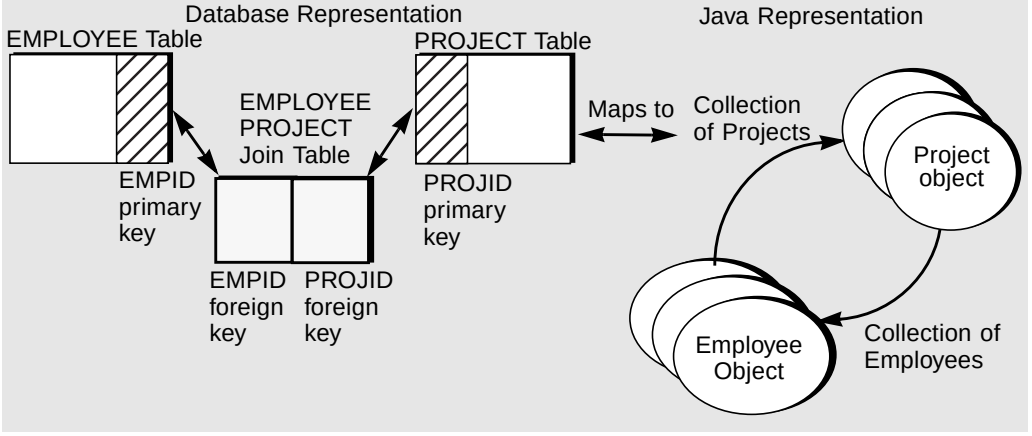

FIGURE 4-3 Foreign Keys and Many-to-Many Relationships

Note – Transparent Persistence does not support duplicate entries in join tables. The many side of the relationship is implemented using `HashSet`, which does not accept duplicate objects.

Managed Relationships

A managed relationship between fields in a pair of classes allows operations on one side of the relationship to affect the other side.

At runtime, if a field in one instance is modified to refer to another instance, the referred instance will have its relationship field modified to reflect the change in relationship.

As described below, Transparent Persistence supports:

- One-one relationships
- One-many relationships
- Many-many relationships

One-One Relationships

With one-one relationships, there is a single-valued field in each class whose type is the other class. Any change to the field on either side of the relationship is handled as a relationship change. If the field on this side is changed from a non-null value to null, then the field on the other side is changed from a non-null value to null. If the field on this side is changed from null to non-null, then the field on the other side is changed to refer to this instance. If the field on the other side had been non-null, then that other relationship is made null before the change is made.

One-Many Relationships

With one-many relationships, there is a single-valued field on the many side and a multi-valued field (collection) on the one side.

If an instance is added to the collection field, the field on the new instance is updated to reference the instance containing the collection field. If an instance is deleted from the collection, the field on the instance will be nullified.

Any change, addition or subtraction of a field on the many side, is handled as a relationship change. If the field on the many side is changed from null to non-null, then this instance is added to the collection-valued field on the one side. If the field on the many side is changed from non-null to null, then this instance is removed from the collection-valued field on the one side.

Many-Many Relationships

With many-many relationships, there are multi-valued, or *collection*, fields on both sides of the relationship. Any change to the contents of the collection on either side of the relationship is handled as a relationship change. If an instance is added to the

collection on this side, then this instance is added to the collection on the other side. If an instance is removed from a collection on this side, then this instance is removed from the collection on the other side.

Note – No warning is given if you delete one object in a managed relationship. Transparent Persistence automatically nullifies the relationship on the foreign key side and deletes the object without asking for confirmation.

You can set the Java Generation options of Transparent Persistence so that managed relationships are generated automatically. You can set these options in the Customize Options pane of the Java Generation wizard (see "Generating Persistence-Capable Classes From a Schema" on page 54), or by choosing Tools > Options, then choosing Java Generation Options under Transparent Persistence (see "Java Generation Options" on page 72).

The following procedure describes how to create a managed relationship when you already have two classes and are taking the "meet-in-the-middle" approach.

▼ To Create a Managed Relationship

1. **Create one relationship field in each of the two classes.**
2. **Ensure that the fields are marked as persistent. (See "To Make a Field Persistent" on page 60.)**
3. **In the Explorer window, expand one of the classes and select its relationship field.**
4. **Open the Properties window for the field.**

   The name of the other class may appear as the value of the Related Class property. If it does not appear, click the property value and then click the ellipsis button (…) to choose a related class. If the class is not persistence-capable, you might need to convert the class (see "Making a Class Persistence-Capable" on page 59).
5. **Choose the other class and click OK.**
6. **Return to the Properties window and click Related Field. Choose the relationship field from the other class.**

   If the field you want does not appear in the drop-down menu, check that it is marked as persistent. If it is already mapped, unmap it using the drop-down menu for its Mapping property.
7. **In the Explorer window, expand the other class and select its relationship field.**
8. **Open the Properties window for the field.**

   Note that the Related Class property and the Related Field properties have been set for you.

   Your two relationship fields now represent a managed relationship.

To map your relationship to a database, see “Mapping Relationships” on page 45.

Developing Persistence-Capable Classes

Capturing a Schema

Before mapping any Java classes to a database schema, you need to capture the schema. Capturing the schema creates a working copy in your file system. This allows you to do your work without affecting the database itself.

Note – It is best to store the captured schema in a package. If you do not have a package to contain the schema, create one by right-clicking on the file system and selecting New Package.

▼ To Capture a Schema

1. **You have three ways to display the Database Schema Wizard:**
   - Right-click on the filesystem and select New > Databases > Database Schema.
   - Choose New from the File menu and then, in the Template Chooser, double-click Databases and select Database Schema.
   - Select Capture Database Schema from the Tools menu.
2. **In the Target Location pane (shown in** FIGURE 4-4**), type a filename for the working copy of your schema, then select a package for the captured schema.**

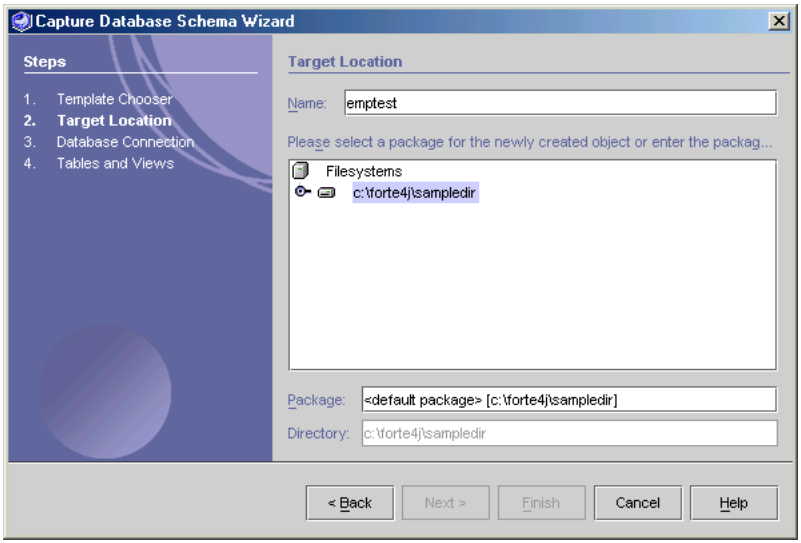

FIGURE 4-4 Database Schema Wizard, Target Location

3. **In the Database Connection pane (shown in** FIGURE 4-5**), if you have a connection established, you can select it from the Existing Connection menu. Otherwise, under New Connection, enter the following information:**

   - The name of the database you are connecting to. (If your database is not listed in the drop-down menu, you might need to quit the wizard and install the driver in the IDE before continuing.)
   - Your system's JDBC driver.
   - The JDBC URL for the database, including the driver identifier, server, port, and database name. For example, `jdbc:pointbase://localhost:9092/sample`.

     The format of a JDBC URL varies depending on which kind of database management system (DBMS) you use—Oracle, Microsoft SQL Server, or PointBase—and the version of that DBMS. Ask your system administrator for the correct URL format for your DBMS.

     FIGURE 4-5 shows the PointBase Server network driver, a server `localhost`, and `port 9092` for a database called `sample`. Your data source might be different.
   - A user name for your database.
   - The password for that user.

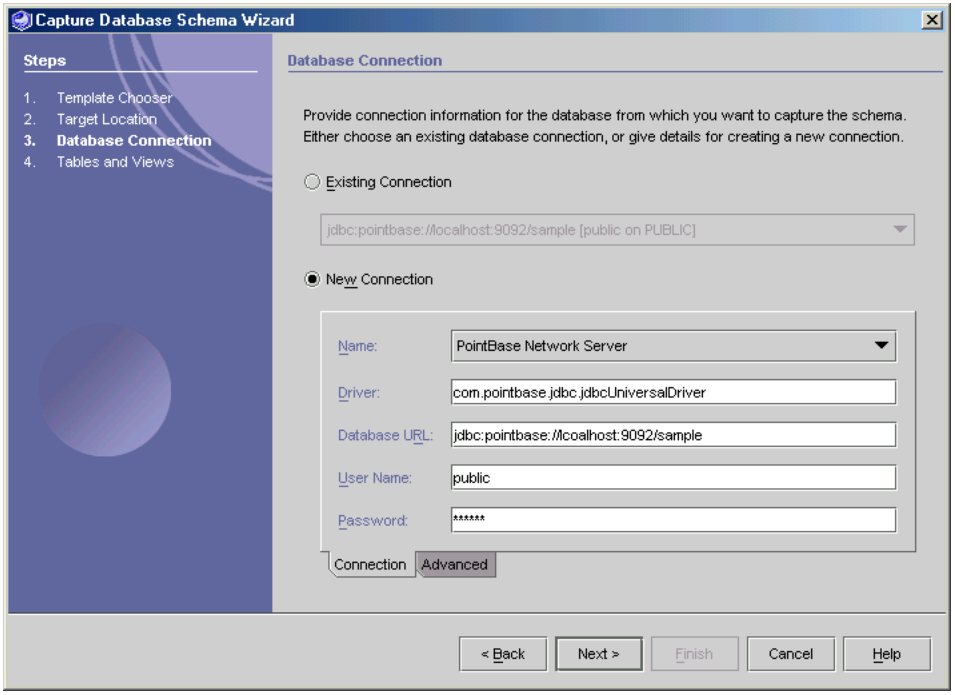

FIGURE 4-5 Database Schema Wizard, Database Connection

4. **In the Tables and Views pane (shown in** FIGURE 4-6**), choose the tables and views you want to capture, then click Finish.**

Note – If you choose one table and exclude another that is referenced to the included table by a foreign key, both tables will be captured even though you specified only one.

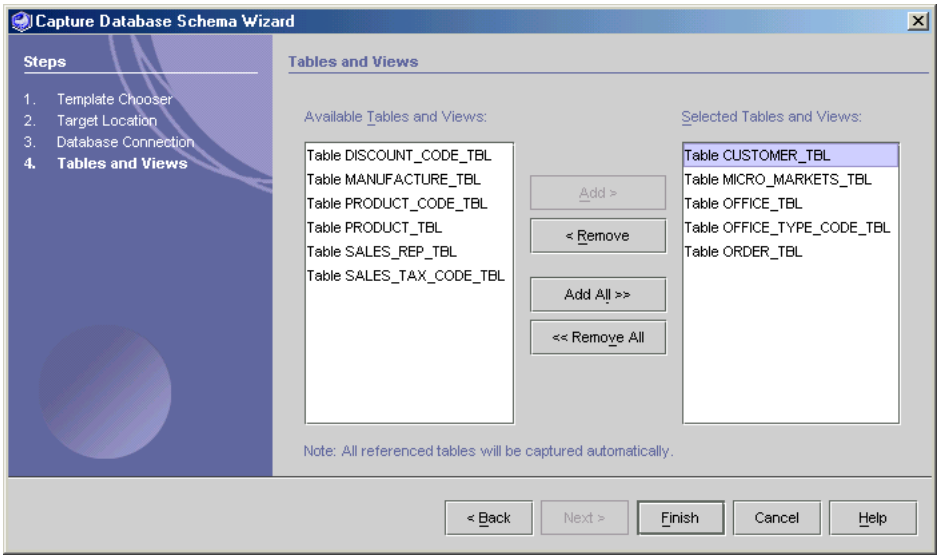

FIGURE 4-6 Database Schema Wizard, Tables and Views

The database and its schema will be represented in the Explorer window, as shown in FIGURE 4-7

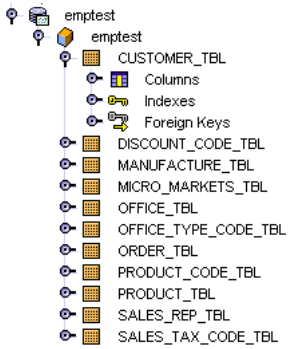

FIGURE 4-7 Database Schema in the Explorer window

Creating Persistence-Capable Classes

Transparent Persistence maps Java classes to tables in a database schema using one of two methods:

- Database to Java mapping

  To generate Java classes from a database schema, see "Generating Persistence-Capable Classes From a Schema" on page 54.

- Meet-in-the-middle mapping

  To create a custom mapping between an existing schema and existing Java classes, see "Mapping Existing Classes to a Schema" on page 59.

Generating Persistence-Capable Classes From a Schema

1. **Select a schema node and choose the Generate Java command. This displays the Generate Java Wizard (see** FIGURE 4-10**), which allows you to:**
   - Choose the target package that will contain your generated Java classes.
   - Customize the options for the classes you are going to generate.
   - Select the database tables for which you will generate corresponding Java classes.
2. **In the Choose Target Location pane (shown in** FIGURE 4-8**), select a package from the packages listed in the dialog window, or enter a new package name in the Package field.**

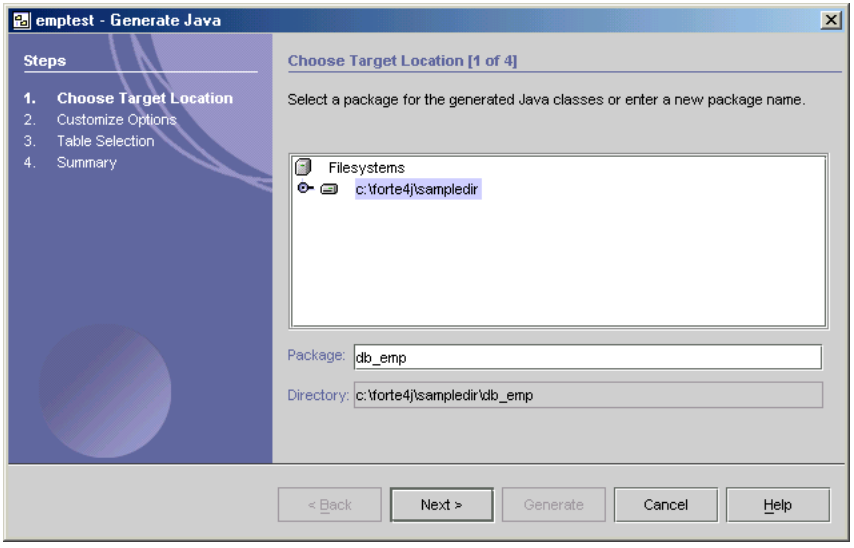

FIGURE 4-8 Java Generation Wizard, Choose Target Location

3. **In the Customize Options pane (shown in** FIGURE 4-9**), select the options for the Java classes you will generate.**

   You can change them for a single session, or save them as default properties for future Java generation sessions. TABLE 4-2 describes the options you can set.

Note – You can change these default options at any time in the Java Generation properties sheet by choosing Tools > Options, then choosing Java Generation Options under Transparent Persistence. See "Java Generation Options" on page 72.

You can set the rules for how relationship fields are named by clicking the ellipsis field (…) in Relationship Naming Policy. See "Relationship Naming Policies" on page 73.

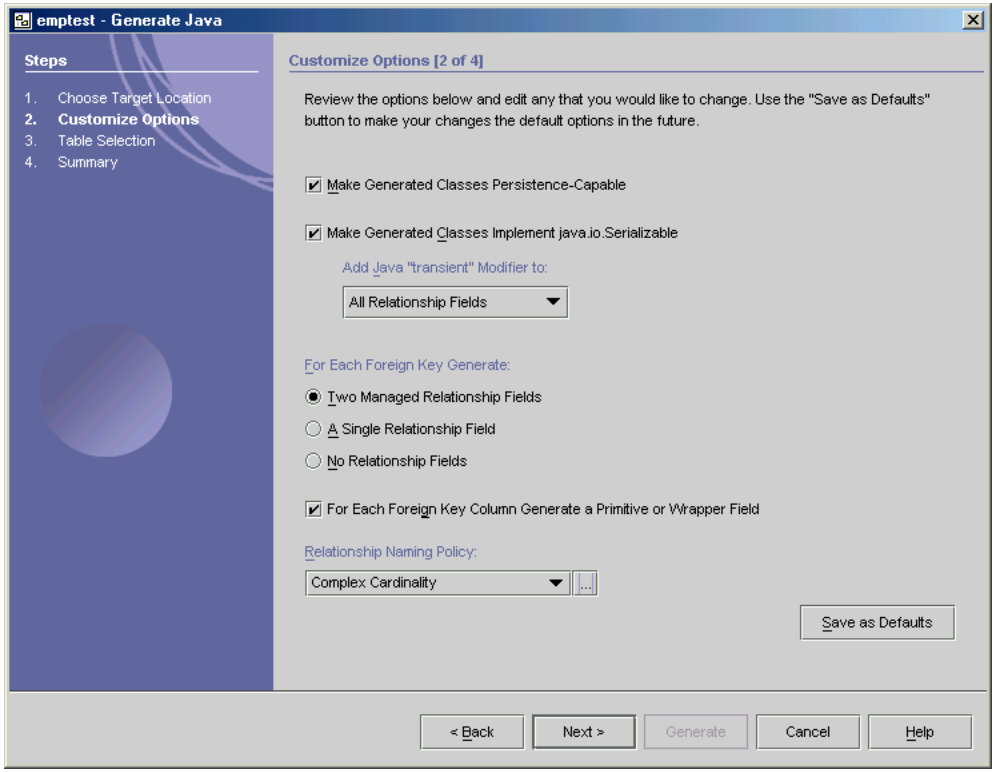

FIGURE 4-9 Java Generation Wizard, Customize Options

4. **In the Table Selection pane (shown in** FIGURE 4-10**), select the tables and views for which you want to generate corresponding Java classes.**

   You can select the tables and views individually, or choose all the tables at once by selecting Add All Tables, or select all views at once by selecting Add All Views.

   Each table is listed under Available. You can use the Add button to specify which tables to map, and edit the class names by clicking on them.

   A listing of `<join table>` in the Java Classes column indicates that there will be a many-to-many relationship between the two classes connected by the join table, but no class is created for the join table itself. If the join table has a primary key, you can create a class for it by clicking on `<join table>` and selecting the class from the drop-down menu, or typing in a class name. This will create a one-to-many relationship between each of the other two classes and the class mapped to the join table. To map the two tables without a relationship, remove the join table from the list.

   Transparent Persistence only generates classes for tables with primary keys. Tables without primary keys are not displayed under Tables Available. Join tables without primary keys appear, but can only link two tables with primary keys, and cannot be used to map classes directly.

   If you want to save classes in different locations, generate the classes for one location, then re-run the wizard, selecting a new location in the Choose Target Location panel.

   Note that you can map multiple classes to the same table or view by running the wizard more than once and customizing the name, or by saving files to different locations.

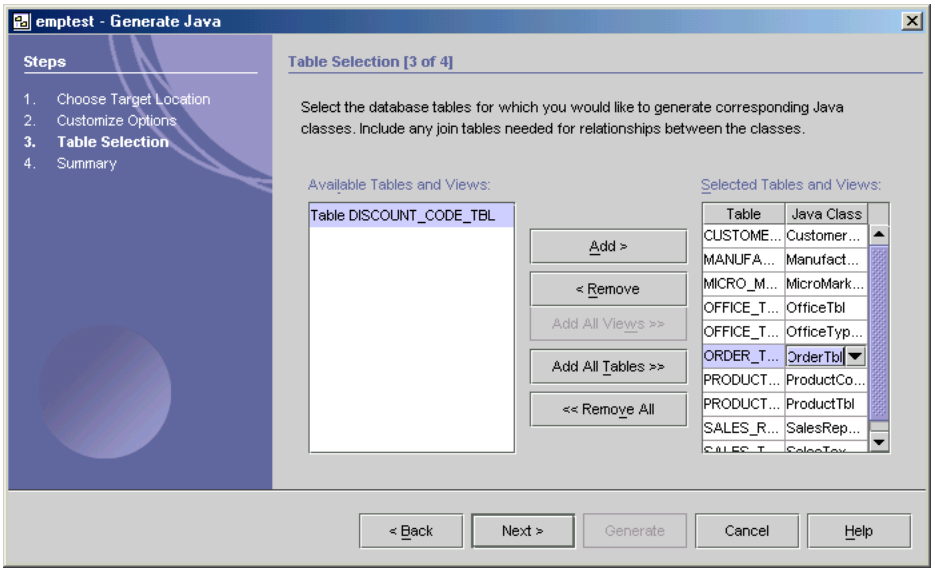

FIGURE 4-10 Java Generation Wizard, Table Selection

Relationship Class Generation

There are many combinations you can choose when selecting tables and views. TABLE 4-1 illustrates the results of each combination. This list assumes two tables, "A" and "B," and a join table "AB."

TABLE 4-1 Relationship Class Generation

A	B	AB	Results
Added	Added	*Join table*	Classes A and B are generated, with two collection relationships, A to B, and B to A.
Added	Added	*Java class name*	Classes A, B, and AB are generated, with four relationship fields created (A to AB, AB to A, B to AB, AB to B).
Added	Added	Not added	A and B are generated, but no relationship fields in A or B are generated, and the AB table is not used at runtime.
Added	Not added	*Join table*	You can not complete the wizard unless you add B or change the name of AB, so you can generate a class for the join table.
Added	Not added	*Java class name*	Class A is generated with a collection relationship to AB and a primitive field for B.
Added	Not added	Not added	Class A is generated with no relationship to B. No relationship fields generated.
Not added	Not added	*Join table*	The join table AB has an incomplete relationship to A and B. You can not generate Java classes until you add both A and B or change the join table name to make it a class.
Not added	Not added	*Java class name*	AB is generated with the foreign keys generated as persisting fields of the primitive types of their foreign key columns.

5. **Click Generate to create a persistence-capable class for each table you selected and map all fields and relationships.**

   If, in the Customize Options pane, you unchecked the Make Generated Classes Persistence Capable check box, non-persistence-capable Java files are generated instead.

   After you have selected the tables and views in the previous panel, the IDE checks for incomplete relationships. All classes that are to be generated are listed on the Summary panel. The panel also displays a list of classes that contain incomplete relationships.

   If you do not map all of the tables in a relationship, the wizard will display warnings or error messages telling you that the relationship will not be mapped. You can use the Previous button to go back and modify your mapping, or click Generate to generate the classes without the relationship.

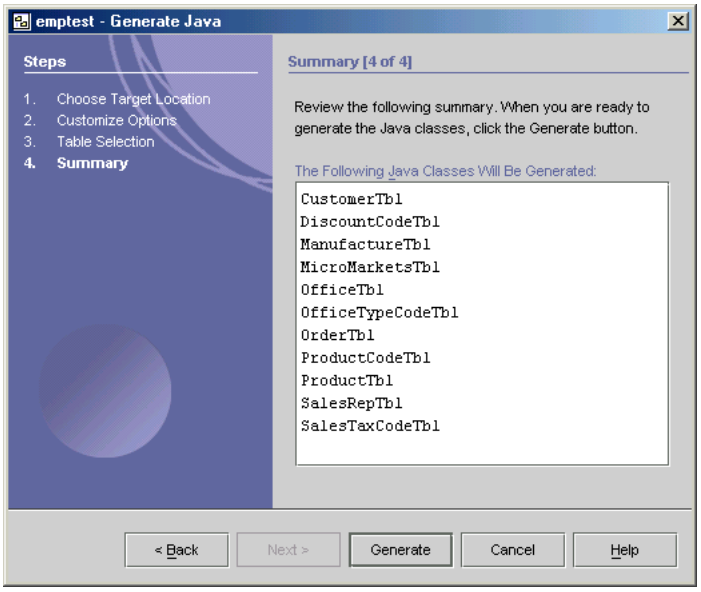

FIGURE 4-11 Java Generation Wizard, Generating Java

If you want to customize your mapped classes, see "Mapping Persistence-Capable Classes" on page 61.

Note – If you want to generate two separate classes mapped to the same primary table, use the Java Generation wizard twice, making sure to rename the generated class. Each of the differently named classes will be mapped to the same table.

Mapping Existing Classes to a Schema

This section discusses how to use Transparent Persistence to customize mappings or to create a mapping for an existing object model.

Before you can map a Java class to a database schema, you must make sure that:

- The database schema is captured and mounted in your Explorer filesystem.

  See "Capturing a Schema" on page 50 for instructions on how to do this.

- Any classes that have relationships to the class you are mapping must be persistence-capable. (The class itself becomes persistence-capable automatically when you start the wizard.)

  See "Making a Class Persistence-Capable" on page 59 for instructions on how to do this.

- All fields that you want to map are marked as persistent.

  See "Making a Field Persistent" on page 60 for instructions on how to do this.

You can edit an existing mapping by returning to the Database Mapping command. The wizard reappears, filled in with all previously set values.

Alternatively, a you can set up or edit a mapping piecemeal by editing the individual properties in the Properties window. All the mapping and persistence information can be accessed through the Properties window, but the wizard provides a way to view and edit groups of classes and fields at one time, providing a useful overview of your mapping model.

Making a Class Persistence-Capable

A class, and all classes related to it, must be persistence-capable before it can be mapped to a database table. The Database Mapping wizard automatically converts your selected class to persistence-capable, but other classes must be converted directly.

You can convert a set of selected classes at once. You should use this approach when converting classes that are related to each other. This makes all relationship fields persistent automatically.

For each class that you want to convert, right-click on the class and select Convert to Persistence-Capable. To convert a group of classes at once (recommended), multi-select the classes by holding the Control key down while choosing the classes. Then right-click and select Convert to Persistence-Capable.

Reverting a Class From Persistence-Capable

Conversely, you can make a persistence-capable class non-persistent by right-clicking on the class and selecting Revert from Persistence-Capable. This will remove all schema mappings and persistent properties from the class.

Note that if you then re-convert the class using Convert to Persistence-Capable, the persistent properties will be restored to their default values, and you will need to map the class to a database schema, as described in "Meet-in-the-middle mapping" on page 45.

Making a Field Persistent

When you make a class persistence-capable, every field that can be interpreted as persistent becomes persistent automatically. If you add any fields, you will need to make them persistent separately if you want to use them to access persistent data.

▼ To Make a Field Persistent

1. **In the Explorer window, expand the class and the Fields node under it and select the field.**

   Persistent fields are displayed with a triangle; relationship fields show a triangle and an arrow; non-persistent fields are displayed with a circle. (See FIGURE 4-12.)

2. **In the Properties window, click on the Persistent property to activate the drop-down menu, then select True.**

   You can make the field non-persistent again by selecting False in the drop-down menu.

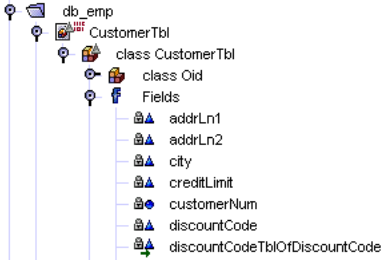

FIGURE 4-12 Persistent Fields

▼ To Map Classes to Tables Using the Database Mapping Wizard

1. **Right-click the class and choose the Map to Database command. This displays the Database Mapping wizard (see** FIGURE 4-13**).**

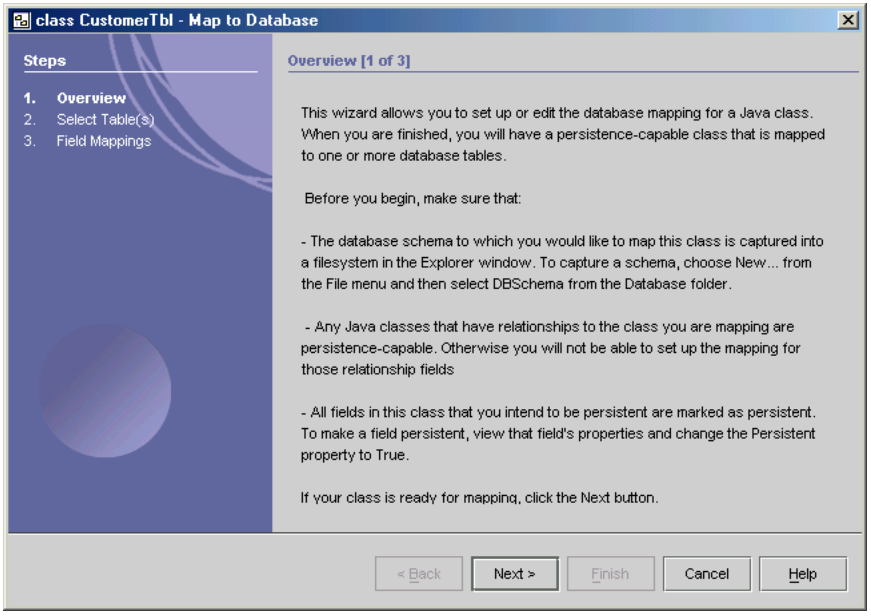

FIGURE 4-13 Database Mapping Wizard Overview

2. **If you have completed the preliminary tasks, click Next to bring up the Select Tables pane of the wizard (see** FIGURE 4-14**). Otherwise, click Cancel, complete the tasks, and restart the wizard.**

3. **Select a primary table from the Primary Table combo box, or click Browse to open the Select Primary Table dialog.**

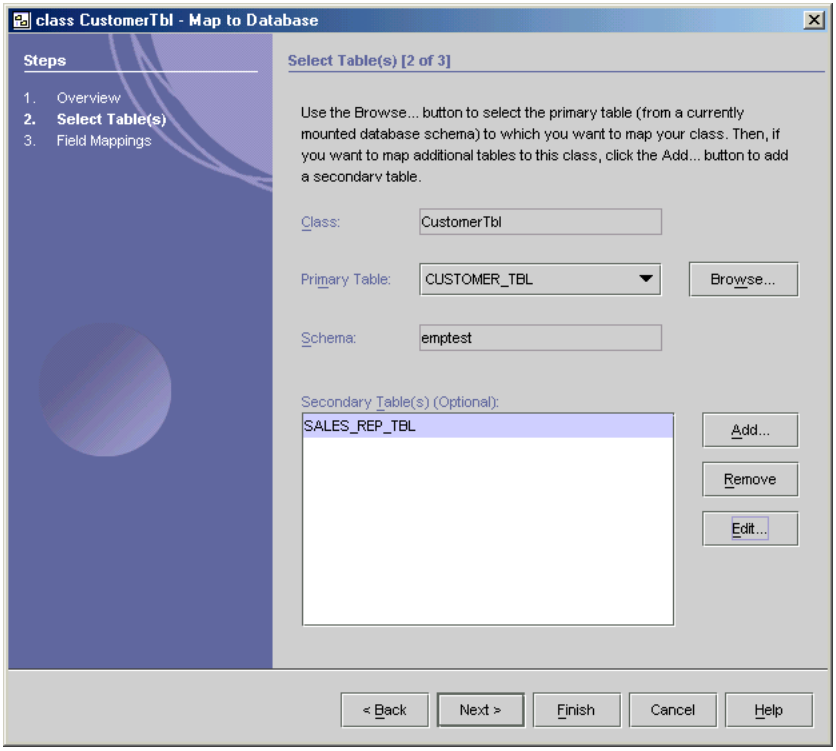

FIGURE 4-14 Database Mapping Wizard, Select Tables

4. **If you open the Select Primary Table dialog (see** FIGURE 4-15**), find a schema and expand it to find its tables. Then select a table and click OK.**

   The table you select as the primary table should be the one that most closely matches your class.

   The table you choose as the primary table must have a primary key, and should be the table that most closely matches the class you are mapping.

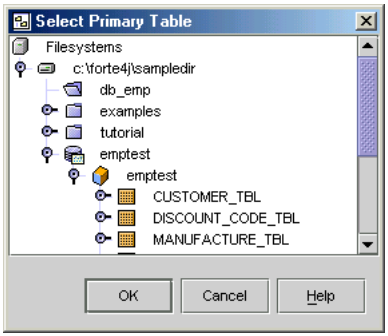

FIGURE 4-15 Select Primary Table Editor

5. **Once the primary table is set up, you can map one or more secondary tables by clicking Add to open the Secondary Table Settings dialog box (see** FIGURE 4-16**).**

   A secondary table enables you to map fields in your persistence-capable class to columns that are not part of your primary table. For example, you might add a DEPARTMENT table as a secondary table in order to include a department name in your Employee class. A secondary table differs from a relationship, in which one class is related to another by way of a relationship field. In a secondary table mapping, fields in the same class are mapped to two different tables. A secondary table enables you to map your field directly to columns that are not part of your primary table. You can use this pane to select secondary tables, and to show how they are linked to the primary table.

   A secondary table must be related to the primary table by one or more columns whose associated rows have the same values in both tables. Normally, this is defined as a foreign key between the tables. When you select a secondary table from the drop-down menu, the wizard checks for a foreign key between the two tables. If a foreign key exists, it is displayed as the reference key by default.

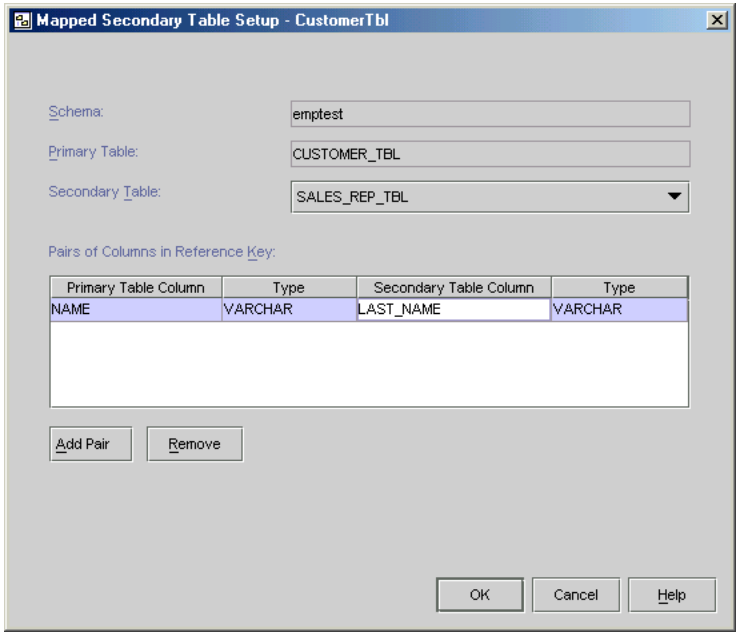

FIGURE 4-16 Mapped Secondary Table Setup

   a. Select a secondary table from the combo box.

      Once you select a secondary table, Transparent Persistence checks to see if there is a foreign key between the primary and secondary tables. If so, the foreign key is displayed as the default reference key. If there is no foreign key, the editor displays “Choose Column,” and you must set up a reference key.

b. To set up a reference key, click <Choose Column> and select a column from the drop-down menu.

Once you pick a primary column, the choices in the secondary column are limited to columns of compatible types. If no column is compatible, the field displays "No Compatible Columns." If you select a primary column that is incompatible with your secondary column, the value of the secondary column reverts to "Choose Column."

If no pair of columns seems to relate in a logical manner, so there can be no logical reference key, you may want to reconsider your choice of a secondary table.

You can select the Add Pair key to set up a complex key using more than one pair of columns.

6. Click OK to save your selections.

7. Click Next in the Database Mapping wizard to bring up the Field Mappings panel of the wizard (see FIGURE 4-17**).**

The Field Mappings panel displays all the persistent fields of the class and their mapping status. You can map a field to a column by selecting the column in the drop-down menu for that field, or try to map all unmapped fields by selecting Automap. Automap will make the most logical selections, ignoring any relationship fields and any fields that have already been mapped. It will not change any existing mappings.

If a field in the class is not listed, it is probably not persistent. This could be because it was added after the class was made persistence-capable or because Persistent was set to False in the Properties window. To make it persistent, click Finish to exit the wizard, then change the field's Properties setting to True.

If you want to map a field to a column from another table that is not available, click Previous to return to the previous wizard page and add a secondary table that contains the column you want.

Unmap works on whatever field or fields are selected. You can unmap a group of fields at once by holding down the Shift key or Control key while selecting the fields you want. If you want to unmap one item, choose `<unmapped>` in the drop-down menu for that field.

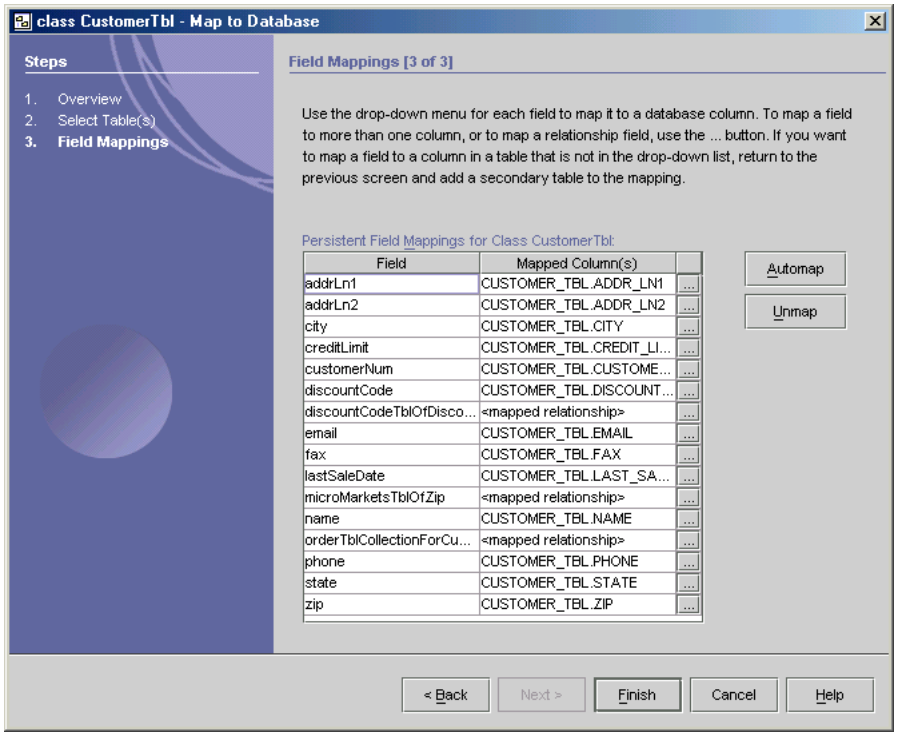

FIGURE 4-17 Database Mapping Wizard Field Mappings

a. To map a field to multiple columns, click the ellipsis button (...) for the appropriate field in the Field Mappings pane to display the Map Field to Multiple Columns dialog box (see FIGURE 4-18**).**

In this dialog box, you add columns to the list of mapped columns. Columns are from the tables you have mapped to this class.You can change the order of the columns by using Move Up/Move Down.

If you do not see the column you want to map, you might need to add a secondary table to your mapping, or change the primary table you have selected. If no columns are listed, you have not yet mapped a primary table, or you have mapped a table that has no columns.

If you map a field to more than one column, all columns will be updated with the value of the first column listed. Therefore, if the value of one of the columns is changed outside of a Transparent Persistence application, the value will only be read if the change was made to that first column. Writing a value to the database overwrites any conflicting changes made to any other columns.

You must also make sure that if you map more than one field to any of these columns, the mappings cannot partially overlap.

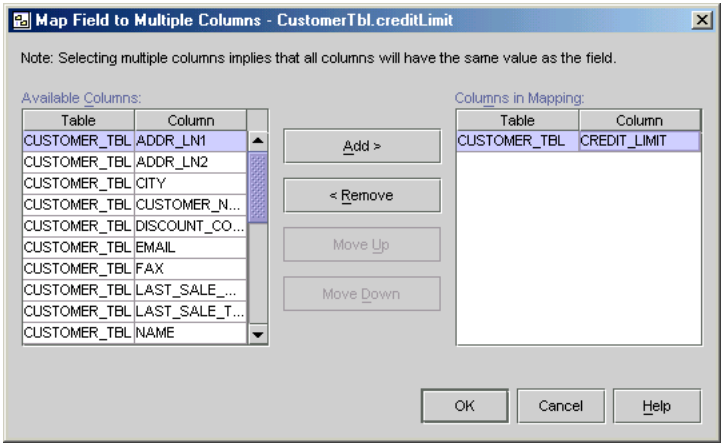

FIGURE 4-18 Map Field to Multiple Columns Dialog Box

Consider the following three examples:

- Field A mapped to Columns A and B, Field B mapped to Column B. Since the mappings only partially overlap, this example will get a validation error at compilation.
- Field A mapped to Column A, and Field B mapped to Column B. Since there is no overlap, this mapping is allowed.
- Field A mapped to Columns A and B, Field B mapped to Columns A and B. Since the mappings completely overlap, this mapping is allowed.

b. Click OK to save the mapping.

Mapping Relationship Fields

When you have foreign keys between database tables, you usually want to preserve those relationships in Java class references. Mapping Relationship Fields lets you specify the relationships that correspond to the class reference fields.

c. To Map a Relationship Field, click on the ellipsis button (…) in the Field Mappings panel next to the drop-down menu of a relationship field to bring up the Relationship Mapping editor (FIGURE 4-19**).**

To use the Relationship Mapping editor outside of the Database Mapping wizard, click on the relationship field in the Explorer and edit its Mapping property.

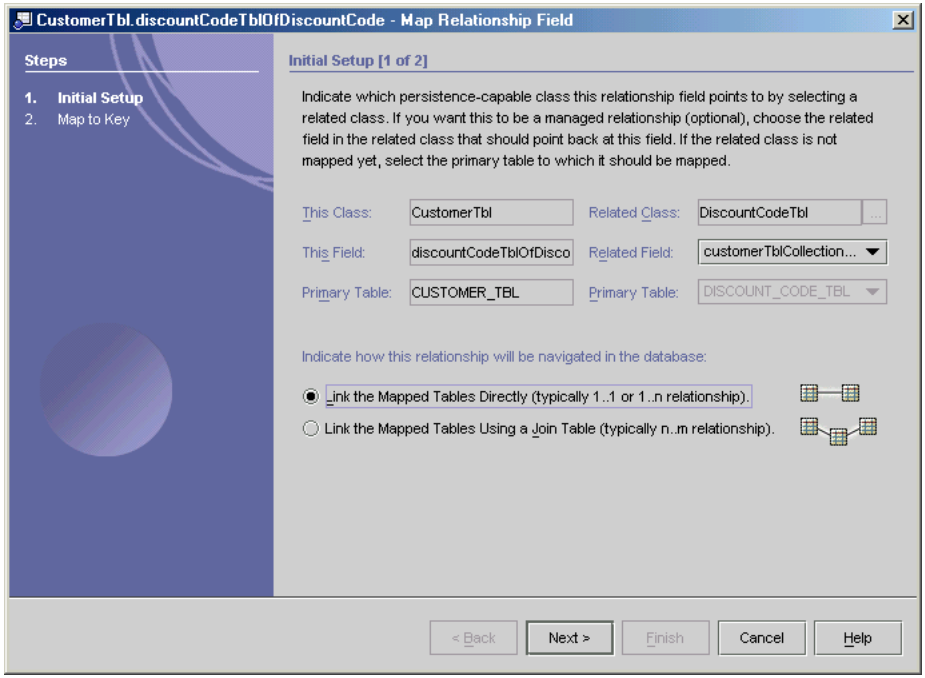

FIGURE 4-19 Relationship Mapping Editor, Initial Setup

- In this pane, verify that the Related Class is set. If the related class is not set, then set it. If the class you want to select is not persistence-capable, you might need to cancel out of the editor, convert the class to persistence-capable, then return.
- Verify that the Related Field (if any) is also correct, and that the Primary Table is set for the related class.

Note – If you have a logical related field, you should choose a Primary Table. That will create a managed relationship.

- Select between linking the tables directly, or through a join table.

d. If your relationships are one-to-one or one-to-many, choose to link the tables directly. Clicking Next opens the Map to Key pane of the Relationship Mapping editor (see FIGURE 4-20**).**

This pane shows:

- An existing mapping if there is one and there were no changes on the initial setup page.
- The default mapping if there is no existing mapping or the mapping is no longer valid.

The editor attempts to determine the most logical key column pairs between the two related classes, based on existing foreign keys. If there are no foreign keys, you need to create the key column pairs by selecting local and foreign columns. The columns in each pair are expected to have the same value.

To create a complex key, use the Add Pair button to add additional Key Column Pairs.

If the Finish button is disabled, you need to choose a key column pair.

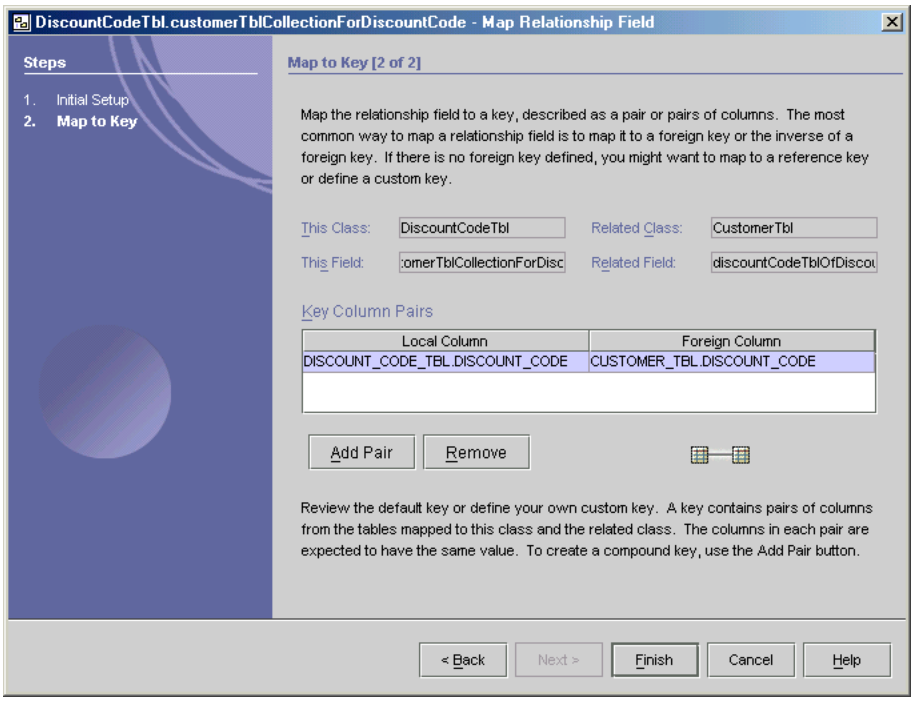

FIGURE 4-20 Relationship Mapping Editor, Map to Key

e. If your relationship is many-to-many, link tables through a join table. Clicking Next opens the Map to Key: Local to Join pane (see FIGURE 4-21**).**

This pane shows:

- The first class and field in the relationship
- The join table to be used to create the relationship between the fields

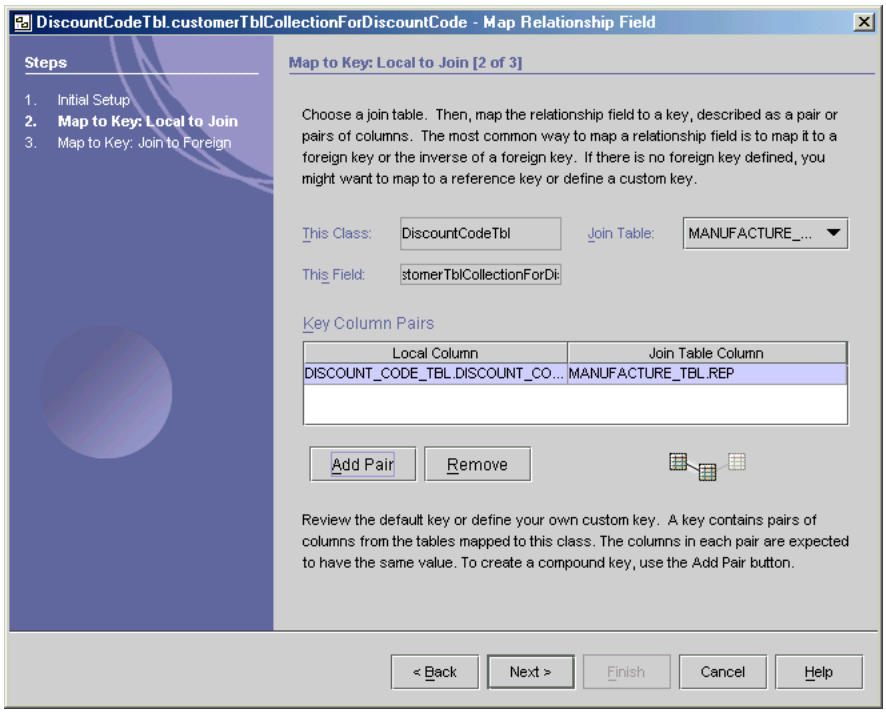

FIGURE 4-21 Relationship Mapping Editor, Map to Key: Local to Join

- Key column pairs between the field join table and the table to which the related class is mapped

  In this pane, you choose a join table, then map the relationship field to a key. This is only the relationship between the table “This Class” is mapped to and the join table. If you don't have a join table, go back to the previous panel and select Link the Mapped Tables Directly.

  Choose a join table that sits between the two tables that your classes are mapped to. The Editor will attempt to determine the most logical key column pairs between the join table and the table that “This Class” is mapped to.

  If the tables have a foreign key between them, the editor will use the foreign key as the default key column pair. If there is no foreign key, then you must create a key by choosing a pair of columns that will allow navigation from the join table to the table to which “This Class” is mapped. The columns in each pair are expected to have the same value.

  To create a compound key, use Add pair to add additional Key Column Pairs.

  If the Next button is disabled, you need to pick a join table or make sure that at least one key column pair exists that has columns on both sides.

f. Click Next to open the Map to Key: Join to Foreign pane.

In this pane, you relate a second table to the join table you chose in the previous pane.

The editor will attempt to determine the most logical key column pairs between the join table and the table that the Related Class is mapped to.

If the tables have a foreign key between them, the editor will use the foreign key as the default key column pair. If there is no foreign key, then you must create a key by choosing a pair of columns that will allow navigation from the join table to the table to which the Related Class is mapped. The columns in each pair are expected to have the same value.

To create a compound key, use Add Pair to add additional key column pairs.

If the Finish button is disabled, you need to choose a valid key column pair.

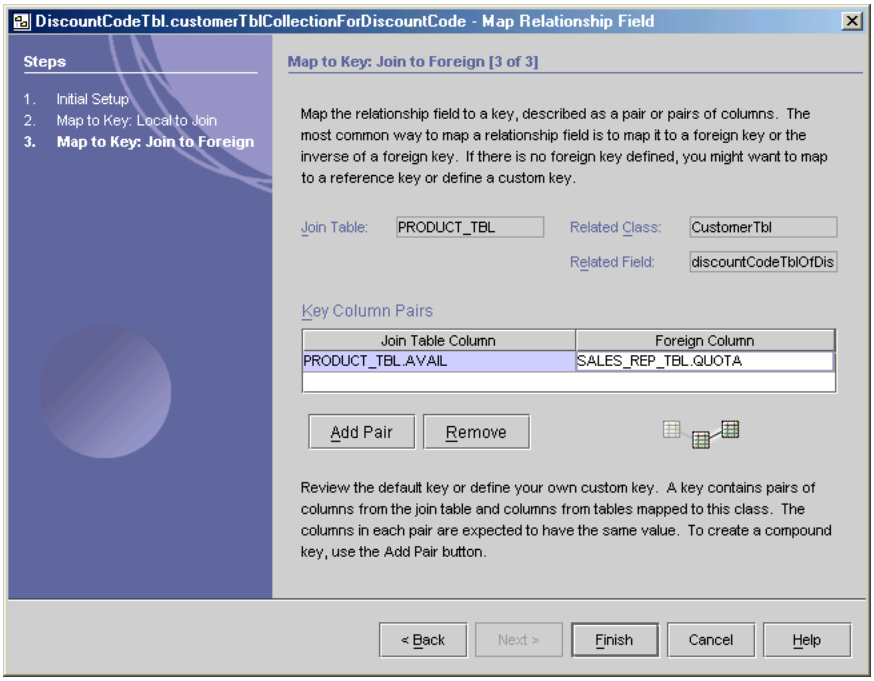

FIGURE 4-22 Relationship Mapping Editor, Map to Key: Join to Foreign

g. Click Finish to return to the Field Mappings pane of the Database Mapping wizard.

8. Click Finish to close the Field Mappings pane and map the Java classes to the database schema.

Setting Options and Properties

Selecting the property sheets of nodes outside the wizards provided by transparent Persistence lets you affect:

- Continuos validation of persistence classes
- Options for Java Generation
- Policies for naming relationship fields
- Properties of persistence-capable classes and fields

Continuous Validation of Persistence Classes

You can open this property sheet by selecting Tools > Options and choosing the Transparent Persistence node.

Setting the Validate Java Changes property to True causes Transparent Persistence to validate changes made in persistence-capable class source code to ensure that they do not cause compilation errors. If a class is modified so that it is no longer valid, a warning dialog appears that gives you three choices:

- OK. Transparent Persistence keeps the change, and makes other changes to the file so that it will not cause a compilation error.
- Undo. Discards the change.
- Ignore. Does nothing. If you choose Ignore, you might encounter difficulties when compiling.

Setting the property to False is the equivalent to selecting Ignore.

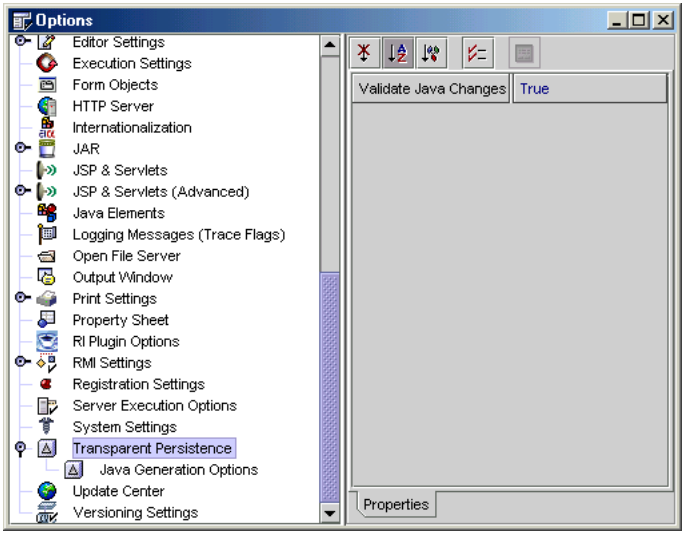

FIGURE 4-23 Validate Java Changes Property

Java Generation Options

Java Generation Options specify the properties that will be used when Java classes and mapping information are generated in the Java Generation wizard. You can override these properties in the second panel of the Java Generation wizard, or by selecting Tools > Options, then choosing Java Generation Options under Transparent Persistence. The Java Generation Properties are described in TABLE 4-2. The property sheet is shown in FIGURE 4-24.

TABLE 4-2 Java Generation Properties

Property	Description
Make Persistence-Capable	Generated Java classes are mapped to a database. If False, you get plain object wrappers for your tables that don't have Transparent Persistence functionality.
Implement Serializable	If True, generated Java classes implement `java.io.Serializable`. This makes the class serializable, so it can be written to a stream between different tiers, such as client and server.
Java Transient Modifier	The transient modifier can be added to certain fields if the class implements `java.io.Serializable`. This property lets you add the `transient` modifier: • Collection Relationship Fields. Single references will be serialized together with the owning object. • All Relationship Fields. No related objects, whether single references or collections, will be serialized with the owner. • No Fields. The complete closure of the objects graph will be serialized.
Primitives for FKs	Whether or not to generate primitive or wrapper fields for each foreign key (FK) column. If you generate relationships as well as primitives fields, there may be implications at runtime.
Relationship Naming	The policy to use to create names for relationship fields. Simple Cardinality provides two rules based on the cardinality of the relationship field. Complex Cardinality provides five rules based on the cardinality of the field and which side of the foreign key it represents. The individual rules are editable. Click on the ellipsis button (…) to open the Relationship Naming Property editor.
Relationship Type	The type of relationship to generate for each foreign key: • Managed Relationship Fields are navigable and updatable from either side of the relationship. • A Single Relationship Field is navigable and updatable only from the class which corresponds to the table containing the foreign key. • If you select None, no relationship fields are generated.

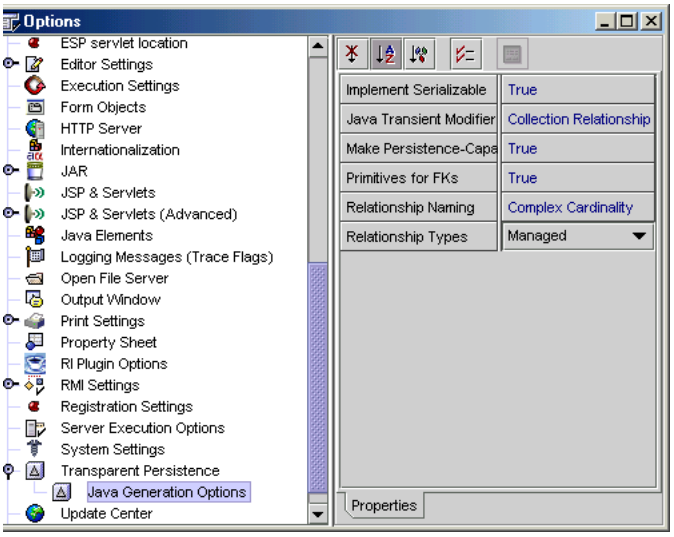

FIGURE 4-24 Java Generation Options

Relationship Naming Policies

When you generate Java classes for tables that have foreign keys, you will create special relationship fields. Because these fields are mapped to pairs of columns in the foreign key, the names for the fields are created by combining the names of the fields. Although it is recommended that you stay with the default settings, you can customize the policies for naming those fields.

You can use the Relationship Naming editor to edit the individual rules for the policy.

▼ To Open the Editor

1. **Select Tools > Options.**
2. **Expand the Transparent Persistence node and then select Java Generation Options.**
3. **Select the editable field for Relationship Naming, and select either Simple Cardinality or Complex Cardinality from the drop-down menu.**
4. **Click the ellipsis button (…).**

   This opens the property editor, as shown in FIGURE 4-25.

Note – You can also open the editor from the second panel of the Java Generation wizard, by clicking on the Relationship Naming Policy field and click the ellipsis button (…).

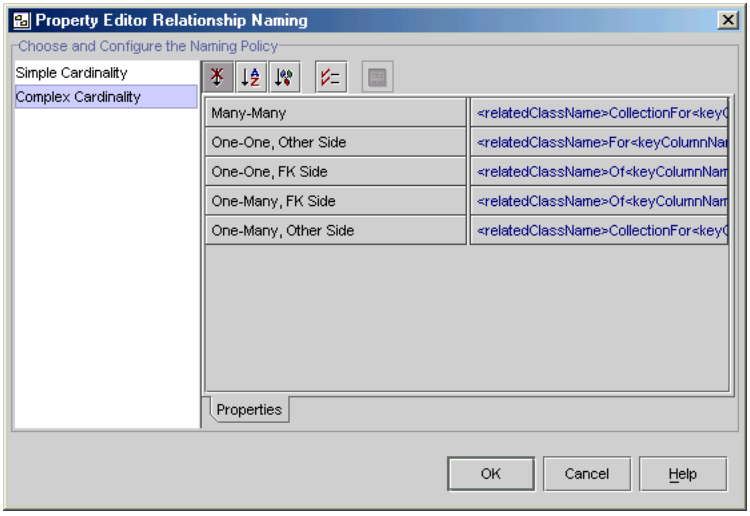

FIGURE 4-25 Relationship Naming Policy Editor

If you select Simple Cardinality, two rules are displayed, as shown in TABLE 4-3.

TABLE 4-3 Simple Cardinality Naming Policy

Rule	Description
Many Side	This is the default name for a field representing a collection relationship
One Side	This is the default name for a field representing a non-collection relationship.

Selecting Complex Cardinality displays five rules, as shown in TABLE 4-4.

TABLE 4-4 Complex Cardinality Naming Policy

Rule	Description
One-Many, FK Side	This is the default name for a non-collection field in a 1:n relationship.
One-Many, Other Side	This is the default name for a collection field in a 1:n relationship.
One-One, FK Side	This is the default name for a field on the foreign key side of a 1:1 relationship.
One-One, Other Side	This is the default name for a field on the non-foreign key side of a 1:1 relationship.
Many to Many	This is the default name for a field in a n:m relationship.

To edit a name, click on the right column and type into the field. To get help with the editing, select the ellipsis button (…) in the field. That opens the Naming Policy Rule Editor, shown in FIGURE 4-26.

Your edits are saved when you click OK.

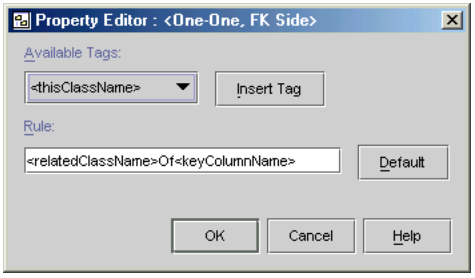

FIGURE 4-26 Naming Policy Rule Editor

To Edit a Naming Policy with the Naming Policy Rule Editor, click in the Rule textbox, then edit the field by inserting a tag from the Available Tags drop-down menu and clicking Insert Tag, or by entering in the text manually.

The tags offered by the drop-down menu are described in TABLE 4-5.

TABLE 4-5 Relationship Naming Tags

Tag	Description
<thisClassName>	Uses the name of the class to which this field belongs.
<relatedClassName>	Uses the name of the "other" class, the class this relationship points to.
<keyColumnName>	Uses the name of the foreign key column or columns.
<thisTableName>	Uses the name of the table that this class maps to.
<relatedTableName>	Uses the name of the table that the related class is mapped to.

Any text typed outside a set of brackets (<>) is treated as a string.

The Editor validates the string before closing. It will warn you if the string is not valid.

Persistence-Capable Class Properties

Persistence-capable classes and persistent fields have several unique properties that can be specified outside of the Database Mapping wizard. TABLE 4-6 describes the properties unique to persistence-capable classes.

TABLE 4-6 Properties for Persistence-Capable Classes

Property	Description
Key Class	An associated class that includes a key field that uniquely identifies a persistence-capable instance. If you use meet-in-the-middle mapping, you must set the Key Class manually. See "Key Fields and Key Classes" on page 81 for more information on setting the Key Class.
Mapped Primary Table	The primary table you select for a persistence-capable class should be the table in the schema that most closely matches the class. You must specify a primary table in order to map a persistence-capable class. See "Mapping Existing Classes to a Schema" on page 59 for information on how to do this.
Mapped Schema	The schema containing the tables to which you are mapping the persistence-capable class. The primary table and any secondary tables must be from this schema. This setting cannot be made until you capture the schema as described in "Capturing a Schema" on page 50.

TABLE 4-6 Properties for Persistence-Capable Classes *(Continued)*

Property	Description
Mapped Secondary Table(s)	Secondary tables let you to map columns that are not part of your primary table to your class fields. For example, you might add a `DEPARTMENT` table as a secondary table in order to include a department name in your `Employee` class. You can add multiple secondary tables, but no secondary table is required. This property is only enabled when Mapped Primary Table is set. See page 63 for more information on adding a secondary table.
Persistence-Capable	Whether the class is persistence-capable or not. This property is only visible when set to true. To convert a class to persistence-capable, see "Making a Class Persistence-Capable" on page 59. To revert a class from perstence-capable, see "Reverting a Class From Persistence-Capable" on page 60.

FIGURE 4-27 shows the properties for a persistence-capable class.

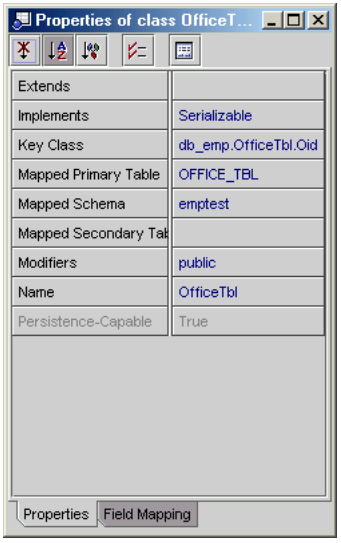

FIGURE 4-27 Persistence-Capable Class Properties

You can unmap a class by choosing `<unmapped>` from the drop-down menu for the Mapped Primary Table property. When you unmap a currently mapped class, a warning appears if there are field mappings or secondary tables. Click OK if you are sure that you want to unmap the class. Otherwise, click Cancel to cancel the mapping status change and leave the class mapped.

Click on the Field Mapping tab at the bottom of the Properties window to see the field mapping properties for a persistence-capable class (FIGURE 4-28).

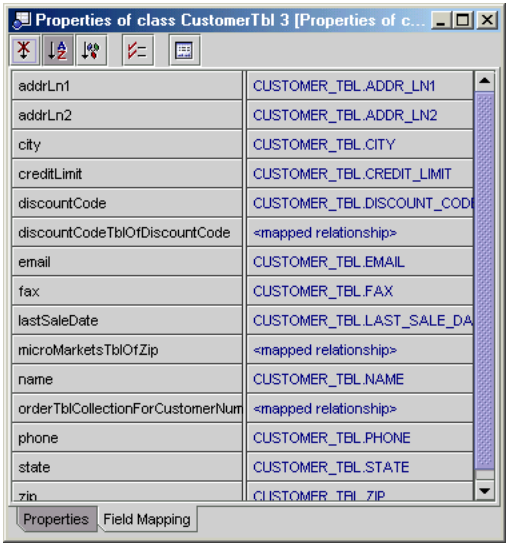

FIGURE 4-28 Field Mapping Properties

Persistent Field Properties

To view the properties of a field, right-click on a field node. A Persistent Field property sheet is shown in FIGURE 4-29.

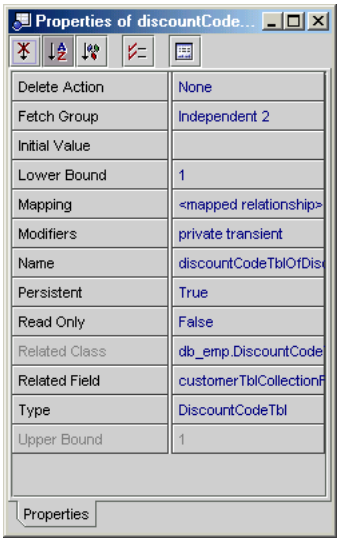

FIGURE 4-29 Persistent Field Properties

You can map a persistent field by choosing a column from the field's drop-down menu in the Mapping property. To map additional columns to that field, click the ellipsis button (…) to display the Map Field to Multiple Columns dialog box. See FIGURE 4-18 for an explanation of the dialog box.

To map a relationship field, selecting the field and clicking the ellipsis button (…) to display the Relationship Mapping editor. See "Mapping Relationship Fields" on page 66 for an explanation of the dialog box.

You can unmap a field by choosing `<unmapped>` or `<unmapped relationship>` from the drop-down menu.

TABLE 4-7 describes the properties unique to persistent fields.

TABLE 4-7 Properties for Persistent Fields

Property	Description
Delete Action (Relationship fields only)	Set to `Cascade` or `None`. `Cascade` indicates that when this field is deleted, all related fields are deleted with it. `None` indicates that only the object represented by this field is deleted.
Related Class (Relationship fields only)	The related class is the class the relationship field points to. For a collection, the related class identifies the type of objects that make up its elements. If a field is not a collection, the property will be disabled.
Related Field (Relationship fields only)	The related fieldcan be set to a relationship field in the related class. Setting this property locks the relationship fields into a managed relationship.
Fetch Group	Specify `Level`, `Independent`, `Default`, or `None`. There are two types of fetch groups, hierarchical and independent. A setting of `Default` for a field means that field will be fetched along with all other fields that have a setting of `Default`. When a field in the Level 1 group is fetched, all fields in group Level 1 and the Default group are fetched as well. Related fields are not allowed to be in any fetch group besides `Default`. Hierarchical groups include the `Default` and `Level` settings, and build on one another (for example, Level 2 includes Level 1 as well). Independent groups include the `Default` group and the specified `Independent` group only (`Independent` 2 does not include `Independent` 1). If the Fetch Group property is disabled, the field is not persistent, not mapped, or is a key field and will always be fetched.
Key Field	If `True`, the field should be mapped to a column in the primary key of the persistence-capable class' primary table.

TABLE 4-7 Properties for Persistent Fields *(Continued)*

Property	Description
Lower Bound (Relationship fields only)	The minimum number of objects a relationship field can hold. The default of 0 means that the field can be null. On the many side of a relationship, this value can be set to any integer value not greater than the Upper Bound. On the one side of a relationship, it can be set to 1 or 0.
Mapping	Shows the mapping status for the field.
Persistent	If `True`, this field's value will be stored in the database.
Read Only	If `True`, this field's value is not updatable to the database.
Upper Bound (Relationship fields only)	The maximum number of objects a relationship field can hold. On the many side of a relationship, this can be set to any integer value, with a default of * (`java.lang.integer.MAX_VALUE`). On the one side of a relationship, the Upper Bound is 1 and cannot be changed.

The field icons in the Explorer change to indicate whether a class or field is persistence-capable. Persistence-capable classes and fields are marked with a triangle, as shown in FIGURE 4-30 and FIGURE 4-31.

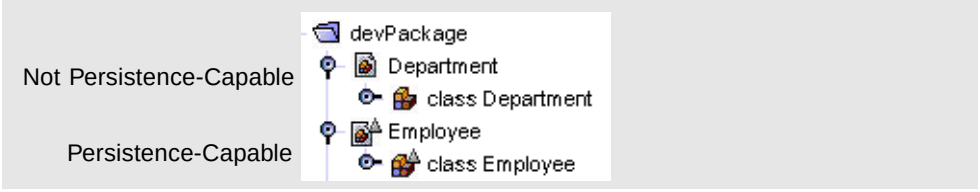

FIGURE 4-30 Class Icons

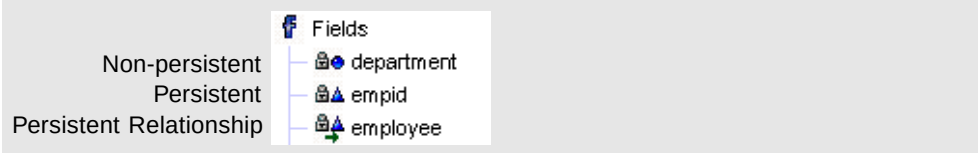

FIGURE 4-31 Field Icons

Key Fields and Key Classes

A Key Class is a class associated with each persistence-capable class that contains unique identifier information for each Transparent Persistence instance. The Java generator creates Key Classes and sets Key Fields automatically. However, if you use meet-in-the-middle mapping, you must set these properties yourself and write the key class.

If you generate Key Classes and Key Fields, then change the fields, you might need to update the oid class. If you create a new class using the Transparent Persistence template, you get a skeleton oid class that you can update.

A Key Class can be either of the following types:

- A static inner class named `Oid`
- A separate class with suffix `Key`

In FIGURE 4-27, the Key Class is set to `db_emp.OfficeTbl.Oid`. This is the `Oid` class set by the Java generator automatically.

▼ To Set up a Key Class and Key Fields

1. **Set the Key Class property on the class node. Make sure the Key Class name is a valid class name.**

2. **Create the Key Class and include all the Key Fields.**

   Each field in the persistence-capable class marked as a primary key must be declared in the Key Class. Each field of the Key Class must have the same name and type as the corresponding field in the persistence-capable class. All fields in the key class must be declared public. The key class must implement `java.io.Serializable`, and override the `equals` and `hashCode` methods.

3. **Set the Key Field property of each field in the persistence-capable class to True for all fields mapped to primary keys. All fields not in the persistence-capable class should be set to False.**

   Following is an example of an inner Oid Class defined for the `Employee` class.

```
public static class Oid {

public long empid;
    public Oid() {
    }
```

```
    public boolean equals(java.lang.Object obj) {
        if(obj==null ||
        !this.getClass().equals(obj.getClass())) return(false);
        Oid o=(Oid) obj;
        if(this.empid!=o.empid) return(false);
        return(true);
    }

    public int hashCode() {
        int hashCode=0;
        hashCode += empid;
        return(hashCode);
    }
```

This next example is a sample Key class defined for the Employees class.

```
public static class EmployeeKey implements java.io.Serializable{

public long empid;
    public EmployeeKey() {
    }

    public boolean equals(java.lang.Object obj) {
        if(obj==null ||
        !this.getClass().equals(obj.getClass())) return(false);
        EmployeeKey = (EmployeeKey) obj;
        if(this.empid!=o.empid) return(false);
        return(true);
    }

    public int hashCode() {
        int hashCode=0;
        hashCode += empid;
        return(hashCode);
    }
}
```

Running an Application

After you compile your application in Forte for Java, you can either add your packages to a `.jar` file or run the application in Forte for Java.

Creating a JAR File

The IDE's JAR packager enables you to create a single JAR (Java ARchive) file from a hierarchy of files, which you can then use in an application outside the IDE. For applications to be able to use Transparent Persistence, both persistence-capable classes and classes that access persistent fields of persistence-capable classes (persistence-aware classes) must be archived by the IDE's JAR packaging tools (for example, JAR, WAR, or EAR packager) to provide the enhancement of the classfiles' byte-code.

When you create a JAR file for persistent classes, you must also take the following into consideration:

- Do not add the Java files to the JAR file, as this can result in unexpected javac errors in future compilations. This can be achieved by having the jarContent node's File Filter property set to all files except `*.java` and `*.jar`. Be careful not to set it to classes only, which would exclude the mapping files (`*.mapping`), which are used by the Transparent Persistence runtime to identify persistence-capable classes.
- Make sure that your schema file (*.dbschema) is included. If the schema file is in the specified package, it will be included automatically. Otherwise, you need to specify the schema files' location in the CLASSPATH.
- When you are using a persistence-aware application outside of the IDE, make sure the following JAR files are included in your CLASSPATH:
  - `.../modules/ext/persistence-rt.jar`
  - `.../modules/dbschema.jar`
  - `.../lib/ext/xerces.jar`
  - `<package>.jar` (`JAR` file with packaged persistence classes)
  - The `JDBC` driver

▼ To Create a JAR File

1. **Open a JAR Packager template using the New From Template wizard.**
2. **Specify the contents of the JAR file.**
3. **Compile the JAR file.**

Note – If you use the JAR Packager to create a `.jar` file for use outside of Forte for Java, you can experience compilation problems unless you accept the default filter of `<all files except *.java and *.jar>`.

For more complete instructions on creating a JAR file, see the Core IDE help topic, "Using the JAR Packager."

Running an Application in Forte for Java

Select the class that contains your application's `main()` method and select Persistence Executor as the value of the Executor property in the Execution tab.

This will invoke the Transparent Persistence enhancer upon class loading, and marks the generated class as implementing the `com.sun.forte4j.persistence.PersistenceCapable` interface. This allows the persistence-capable classes to interact with the runtime environment.

The `com.sun.forte4j.persistence.PersistenceCapable` interface declares a set of methods that allows users of persistence-capable classes (application developers) to check and reset the status of instances of these classes.

Neither the developer of the classes nor the application developer who uses them needs to be aware of what is in the generated byte code. The class developer can concentrate on developing an accurate model of the persistent data.

If you don't intend to use the Persistence Executor, for instance, to execute the persistence-capable classes outside of the Forte for Java IDE, you have to archive your persistence-capable and persistence-aware classes using the IDE's jar packager. This ensures that the enhancement is applied to the classes while they get archived. This step requires that the Transparent Persistence module is enabled in Forte for Java.

Note – When using persistence-capable classes within a web module for JSP/Servlet applications, package the persistence-capable classes as a JAR file and put the JAR into the web-module's `web-inf/lib` directory. Do not put the persistence-capable classes directly into the web module's `web-inf/classes` directory unless you intend to create a WAR file to deploy the web application. The Transparent Persistence classfile enhancement only takes place with the IDE's archiving tools (for examples, a JAR or WAR packager) or if the Persistence Executor is used.

Supported Data Types

Transparent Persistence supports a set of JDBC 1.0 SQL data types that are used in mapping Java data fields to SQL types. TABLE 4-8 lists these data types and notes whether each type is supported.

TABLE 4-8 Supported Data Types

JDBC SQL Data Type
BIGINT
BIT
CHAR
DATE
DECIMAL
DOUBLE
FLOAT
INTEGER
LONGVARCHAR
NUMERIC
REAL
SMALLINT

TABLE 4-9 lists the nullability of supported data types.

TABLE 4-9 Data Type Conversions in Mappings

Java Type	JDBC Type	Nullability
boolean	BIT	NON NULL
java.lang.Boolean	BIT	NULL
byte	TINYINT	NON NULL
java.lang.Byte	TINYINT	NULL
double	FLOAT	NON NULL
java.lang.Double	FLOAT	NULL
double	DOUBLE	NON NULL

TABLE 4-9 Data Type Conversions in Mappings *(Continued)*

Java Type	JDBC Type	Nullability
java.lang.Double	DOUBLE	NULL
float	REAL	NON NULL
java.lang.Float	REAL	NULL
int	INTEGER	NON NULL
java.lang.Integer	INTEGER	NULL
long	BIGINT	NON NULL
java.lang.Long	BIGINT	NULL
long	DECIMAL (scale==0)	NON NULL
java.lang.Long	DECIMAL (scale==0)	NULL
long	NUMERIC (scale==0)	NON NULL
java.lang.Long	NUMERIC (scale==0)	NULL
short	SMALLINT	NON NULL
java.lang.Short	SMALLINT	NULL
java.math.BigDecimal	DECIMAL (scale!=0)	NON NULL
java.math.BigDecimal	DECIMAL (scale!=0)	NULL
java.math.BigDecimal	NUMERIC	NULL
java.math.BigDecimal	NUMERIC	NON NULL
java.lang.String	CHAR	NON NULL
java.lang.String	CHAR	NULL
java.lang.String	VARCHAR	NON NULL

Note – Transparent Persistence does not support BLOBs as mapped column types. To fetch or update BLOBs, you need to use separate JDBC transactions.

CHAPTER 5

Developing Persistence-Aware Applications

This chapter describes the Transparent Persistence runtime environment and illustrates how to use it to perform persistence operations. It also addresses Transparent Persistence programming issues.

The Transparent Persistence API controls interaction with the database. Applications use the API to establish a connection to a specific database and create transactions. Insert and delete must occur within the context of a transaction.

Overview

The Transparent Persistence runtime environment gives Java developers a consistent interface to persistent data, by translating instances of persistence-capable classes and methods of the Persistence Manager into instructions for the particular database that the application is using.

You can view the runtime environment with several Java interfaces. These interfaces provide a set of persistent data methods that provide the functionality for translating method calls into instructions to a specific database.

After persistence-capable classes are mapped to a schema, you can access persistent data by calling methods of the persistence-capable classes and the persistence-aware runtime support classes. You use Forte for Java's regular editing, compiling, test run, and deploying facilities to write code that uses persistence-capable classes.

The Transparent Persistence implementation of the runtime classes is defined by the `com.sun.forte4j.persistence` interfaces. Transparent Persistence includes a file called `persistence-rt.jar` that has implementations of these interfaces.

Transparent Persistence applications perform the standard steps for database interaction with Java method calls, without using a query language or writing Java code specific to a given database. The standard steps include: connecting to the database; starting a transaction; selecting, inserting, updating, or deleting persistent data; then committing (or rolling back) the transaction.

When an application loads data from the database, it uses instances of the persistence-capable classes that model the data. If the application changes the value of a persistent field, the Transparent Persistence runtime environment tracks that change and saves the new value into the database when the application commits its transaction. When an application needs to get data, the developer calls methods of a Persistence Manager (which returns instances of persistence-capable classes). When it needs to change data, it calls methods of the persistence-capable instances, and so on.

The sections that follow describe the ways in which applications can create and use instances of persistence-capable classes.

Developing Persistence-Aware Classes

Write your application in the Java programming language. Use whatever existing classes you need and create your own Transparent Persistence objects and classes just as you would use any other Java object or class. The only difference between these objects and classes is that the persistent Transparent Persistence objects save their data in the database. Thus, you do not need to know whether data is from the database, local variables, or other sources.

Persistence-Aware Logic

FIGURE 5-1 shows a typical architecture for using Transparent Persistence in a real-world application. The application conforms to a standard J2EE architecture and features a JSP or servlet component that manages some interaction with end users in remote locations. The JSP or servlet processes end-user input, determines what action is required, and then calls on a middle-tier service to carry out that action. If the end user wants to see an employee record, the JSP or servlet should be able to call on a middle-tier service that will return the employee records, without needing to know how that record is obtained. In other words, the JSP or servlet should not contain persistence-aware logic.

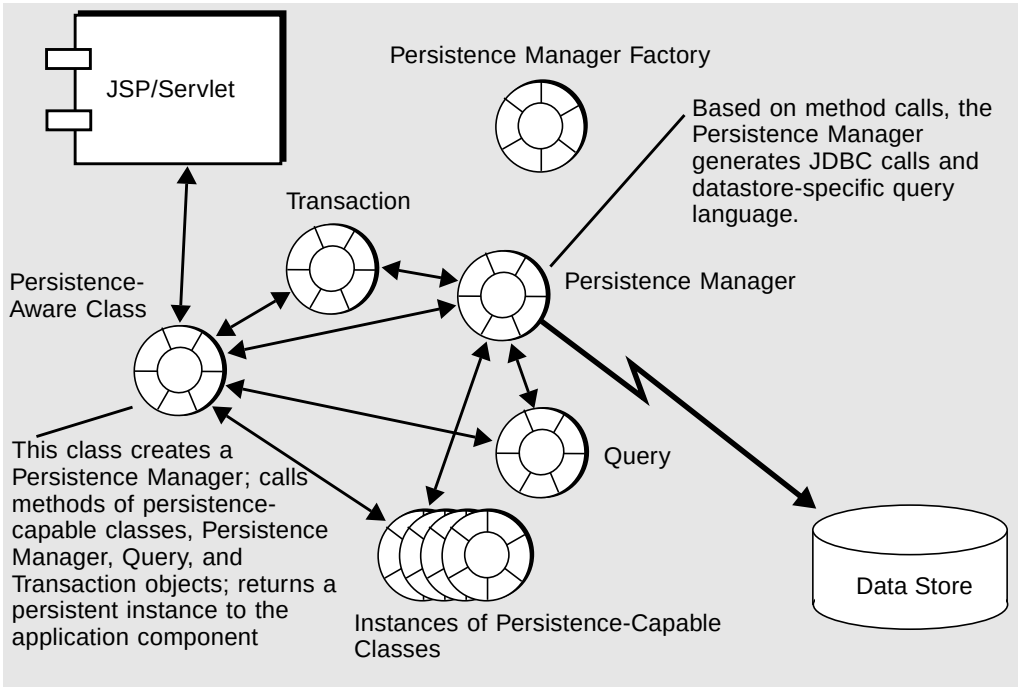

FIGURE 5-1 Moving Persistence-Aware Logic to Its Own Class

To achieve this, the persistence-aware logic has been moved to a separate class. The JSP or servlet can request an employee record by calling a method of the persistence-aware class using an approach like the following:

```
Employee requestedEmployee =
PersistentAwareInstance.getEmployeeData("485843");
```

The persistent-aware instance can then perform all the operations necessary to obtain an `Employee` instance for the employee record that was specified and return it to the JSP/servlet. The persistent instance remains associated with the Persistence Manager and its transaction, even after the persistence-aware class has passed it to the JSP/Servlet. This means that the JSP/servlet can update field values, and the Persistence Manager will automatically generate a database update operation, and manage it in accordance with current transaction and concurrency strategy.

If the end user supplies data for a new employee record, the JSP/servlet can create a new instance and pass it to the persistence-aware class:

```
Employee newEmployee = new Employee(<data>);
PersistentAwareInstance.addEmployeeData(newEmployee);
```

The persistent-aware class can handle it like this:

```
PersistenceManager.makePersistent(newEmployee);
```

In the architecture shown in FIGURE 5-1, a JSP/servlet handles multiple end users concurrently. It maintains a separate session for each user, and a session may include a sequence of HTTP requests exchanged between the end user's web browser and the JSP/servlet. When the JSP/servlet calls on the persistence-aware instance for database services, the persistence-aware instance must be able to track which JSP/servlet sessions initiated the request, and keep all requests from a single session isolated from those of other sessions.

A Persistence Manager generally manages a set of TP instances created or fetched in multiple data store operations, so it is capable of managing persistent instances generated by a conversational session.

Development Steps

An application developer using Transparent Persistence classes uses methods of Transparent Persistence classes and runtime environment objects to work with data. This section summarizes the basic sequence of method calls.

1. Create or obtain a Persistence Manager Factory.

   The Persistence Manager Factory is a configurable component, with properties that hold database connection information. You might already have a Persistence Manager Factory that has been configured in your environment and is accessible using JNDI lookup. See "Creating a Persistence Manager Factory" on page 92 for more information.

2. (Optional) Create a Connection Factory.

   This is necessary only if you want to implement connection pooling. See "Pooled Connections" on page 97 for more information on this approach.

3. Create a Persistence Manager.

   Each session will generally create its own Persistence Manager. Unless the application overrides it, the Persistence Manager will use the connection defined by the properties of the Persistence Manager Factory. See "Creating a Persistence Manager" on page 97 for more information.

4. Access the transaction from the Persistence Manager by calling `currentTransaction()`.

   In most cases, the application begins a transaction. The transaction object is obtained from the Persistence Manager, and applies to instances managed by the Persistence Manager. See "Transactions" on page 101 for more information.

5. Use the `Query` interface to access instances of persistence-capable classes from the database.

   Modify the instances by calling their methods. If you want to insert or delete instances, use the appropriate methods on the `PersistenceManager` interface.

   As the application queries the database, modifies records, and adds new records, it will create a set of persistent instances that represent the data it needs. The Persistence Manager manages all the database interactions for this set of instances. In other words, the set of persistent instances managed by one Persistence Manager will be the session's view of the data.

6. Commit or abort the transaction.

   Commit the transaction to save your updates to the database; abort (roll back) the transaction to leave the database as it was before your transaction began.

   When the application commits the transaction, Transparent Persistence performs all database interactions indicated by the current status of each persistent instance. If there are instances that were made persistent during the transaction, Transparent Persistence will generate inserts; if there are instances that were deleted during the transaction, it will generate deletes; if there are instances that were updated during the transaction, it will generate updates.

7. Perform additional transactions.

   You can reuse the same Persistence Manager instance for additional transaction, or you can use a different Persistence Manager instance.

8. Close the Persistence Manager and exit the application.

FIGURE 5-2 presents these steps in a flowchart.

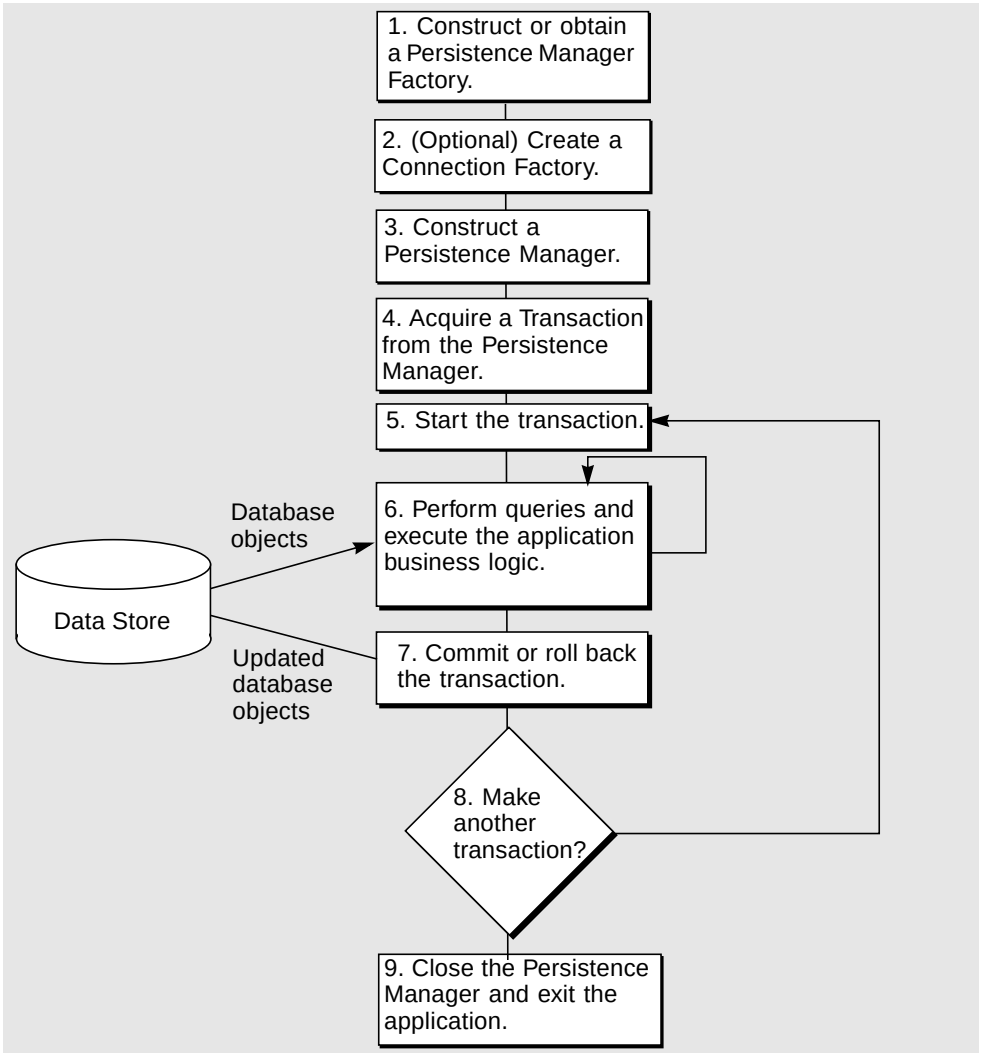

FIGURE 5-2 Transparent Persistence Application Logic

Creating a Persistence Manager Factory

The basis for a persistence-aware application is the Persistence Manager Factory. The Persistence Manager Factory is implemented as a class that developers can instantiate directly. Other objects are obtained by calling the appropriate methods of the Persistence Manager Factory or the Persistence Manager. In many cases, a

developer starts with a Persistence Manager Factory that has already been configured in the environment and can be located through JNDI calls. In that case, the developer can skip to "Creating a Persistence Manager" on page 97.

The standard way for the application to acquire a connection is through the Persistence Manager Factory. The Persistence Manager Factory's configurable properties include the values used to connect to a database. The application instantiates and configures the Persistence Manager Factory, then creates a Persistence Manager that will use the connection information configured into the Persistence Manager Factory.

Create a persistence-aware class by selecting New > Classes > Class. Give the class a name and click Finish.

TABLE 5-1 discusses each method in detail.

TABLE 5-1 `PersistenceManagerFactory` Methods

Method	Description
`setOptimistic` `getOptimistic`	The transaction mode that specifies concurrency control. The default is true.
`setRetainValues` `getRetainValues`	The transaction mode that specifies the treatment of persistent instances after commit. The default is true.
`setIgnoreCache` `getIgnoreCache`	The query mode that specifies whether cached instances are considered when evaluating the filter expression. This is always true. Changing to 'false' throws `JDOUnsupportedOptionException`
`setNontransactionalRead` `getNontransactionalRead`	The Persistence Manager mode that allows nontransactional instances to read outside of a transaction. The default is true.
`setConnectionFactory` `getConnectionFactory`	The connection factory from which database connections are obtained.
`setConnectionMinPool` `getConnectionMinPool`	Minimum number of connections in the connection pool
`setConnectionMaxPool` `getConnectionMaxPool`	Maximum number of connections in the connection pool
`setConnectionFactoryName` `getConnectionFactoryName`	The name of the Connection Factory from which database connections are obtained. This name is looked up with JNDI to locate the Connection Factory.
`setConnectionTransactionIsolation` `getConnectionTransactionIsolation`	Chooses a nondefault isolation level. The `level` argument is any of the `java.sql.Connection.TRANSACTION_*` options supported by the underlying database.

TABLE 5-1 `PersistenceManagerFactory` Methods *(Continued)*

Method	Description
`getPersistenceManager`	Returns a Persistence Manager instance with the specified properties. The default values for option settings are set to the value specified in the Persistence Manager Factory before returning the instance. After the first use of getPersistenceManager, none of the set methods will succeed.
`getProperties`	Transparent Persistence stores certain nonoperational properties and make those properties available to the application using a Properties instance. This method retrieves the Properties instance. Each key and value is a String. The keys required for this implementation are: VendorName: The name of the vendor. VersionNumber: The version number string. Any Persistence Manager Factory property settings become the default settings for Persistence Managers created by the factory and, after a Persistence Manager is created, the Persistence Manager Factory can no longer be changed.
`QueryTimeout` `UpdateTimeout`	This method avoids deadlocks in the database by waiting a specified number of seconds for the completion of the query or update associated with this instance of the Transaction before timing out. The value is stored in seconds; zero means unlimited. It is the default for all Transactions to the underlining database. Persistence Manager Factory settings cannot be changed after creation of the first Persistence Manager. Transaction timeout can be changed as needed. PointBase does not currently support `PreparedStatement.setQueryTimeout()`. Add `,locks.timeout=value` to the URL or `pointbase.ini` file to use any other than default value (current default value is set to 60 seconds). However, be aware that `locks.timeout=0` sets the timeout to 0 seconds, rather than the `setQueryTimeout(0)` behavior of setting it to unlimited. `Locks.timout` is set on the server side, not the client side. This means the value will hold for all connections.

Connecting to Databases

Connections are opened and managed by the Transparent Persistence runtime environment. The Persistence Manager Factory is a configurable component, and its configurable properties include the values used to connect to a database. The resulting Persistence Manager uses the connection information that was configured

into the Persistence Manager Factory, such as the database's URL and a valid user name and password for the database. When the application first performs an operation that requires a connection, such as submitting a query for execution, the Persistence Manager opens a connection.

There are four connection management scenarios:

- Simple connection
- Pooled connections
- Distributed transactions
- Managed connections

In a non managed environment (simple and pooled connections), transaction completion is handled by the Connection that is managed internally by the Transaction. In the managed environment, transaction completion is handled by the `XAResource` associated with the Connection. In both cases, the Persistence Manager implementation is responsible for setting up the appropriate interface to the Connection infrastructure.

Connection Factory

For implementations that layer on top of standard Connector implementations, the configuration typically supports all of the associated Connection Factory properties. You can configure the Connection Factory directly or through the Persistence Manager Factory.

TABLE 5-2 discusses each method in detail.

TABLE 5-2 `ConnectionFactory` Methods

Method	Description
`URL`	URL for the data source.
`UserName`	Name of the user establishing the connection.
`Password`	Password for the user.
`DriverName`	Driver name for the connection.
`ServerName`	Name of the server for the data source.
`PortNumber`	Port number for establishing connection to the data source.
`MaxPool`	Maximum number of connections in the connection pool.
`MinPool`	Minimum number of connections in the connection pool.
`MsWait`	Number of milliseconds to wait for an available connection from the connection pool before throwing an exception.

TABLE 5-2 `ConnectionFactory` Methods *(Continued)*

Method	Description
`LogWriter`	PrintWriter to which messages should be sent.
`LoginTimeout`	Number of seconds to wait for a new connection to be established to the data source.
`TransactionIsolation`	Transaction isolation level for all connections.

Simple Connections

In the simplest case, the Persistence Manager directly connects to the database and manages transactional data. In this case, there is no reason to expose any Connection properties other than those needed to identify the user and the data source. During transaction processing, the Connection is used to satisfy data read, write, and transaction completion requests from the Persistence Manager.

If the application does not require pooled connections, only the following properties of the `PersistenceManagerFactory` need to be configured:

- `ConnectionUserName`–Name of the user establishing the connection
- `ConnectionPassword`–Password for the user
- `ConnectionURL`–URL for the data source
- `ConnectionDriverName`–Driver name for the connection

These will become the default values for any Persistence Manager instances created by that Persistence Manager Factory.

For example, the constructor might initialize a Persistence Manager Factory as follows:

```
public DataSource() {
PersistenceManagerFactory pmf = new
PersistenceManagerFactoryImpl();
    pmf.setConnectionUserName("scott");
    pmf.setConnectionPassword("tiger");

pmf.setConnectionDriverName("oracle.jdbc.driver.OracleDriver");
    pmf.setConnectionURL("jdbc:oracle:thin:@DIESEL:1521:ORCL");
    setOptimistic(false); // It is true by default.
}
```

Pooled Connections

In a slightly more complex situation, the Persistence Manager Factory creates multiple Persistence Manager instances that use connection pooling to reduce resource consumption. The Persistence Managers are used in single database transactions. In this case, a pooling Connection Factory is a separate component used by the Persistence Manager instances. The Persistence Manager Factory will include a reference to the connection pooling component, either as a JNDI name or as an object reference. The connection pooling component is configured separately, and the Persistence Manager Factory needs to be configured to use it.

If any other connection properties are required, then you must configure `setConnectionMinPool` and `setConnectionMaxPool` in the Persistence Manager Factory.

During the execution of a session's business method, running a long-duration optimistic transaction, a connection might be required to fetch data from the database. The Persistence Manager requests a connection from the connection pool to satisfy the request. Upon completion of the request, the connection is returned to the pool.

In a database transaction, `Transaction` keeps the acquired connection for the duration of the session. After completion of the session (either commit or rollback), the connection is returned to the pool and reused for a subsequent transaction.

Creating a Persistence Manager

The Persistence Manager is the starting point for the application's interaction with the Transparent Persistence runtime environment. It encapsulates information about a specific database, opens a connection, and manages queries and transactions. A Persistence Manager Factory must be configured before you can declare a Persistence Manager.

In a persistence-aware class, declare a Persistence Manager and create a Persistence Manager instance:

```
private PersistenceManager pm;
this.pm = pmf.getPersistenceManager();
```

Each Persistence Manager supports one transaction at a time, and this transaction applies to all of the transactional instances of persistence-capable classes that it creates. To work with the transaction, the application obtains a transaction object from the Persistence Manager:

```
Transaction myTx = myPersistenceManager.currentTransaction();
```

In most cases, the application will be running local transactions from a single database. The application starts and completes these transactions by calling Transaction object methods:

```
myTx.begin();
myTx.commit(); // or myTx.rollback();
```

The Persistence Manager normally manages all interactions with the database, including refreshing cached copies of persistent data, and the application only needs to identify transaction boundaries.

TABLE 5-3 discusses each method in detail.

TABLE 5-3 `PersistenceManager` Methods

Method	Description
`isClosed`	Returns false upon construction of the Persistence Manager instance. Returns true only after the `close` method completes successfully.
`close`	Verifies that the Transaction is not active. Otherwise, it throws an exception. Releases all resources (e.g., `Transaction`). After the `close` method completes, all Persistence Manager methods except `isClosed()` throw an exception.
`currentTransaction`	Returns the Transaction instance associated with the Persistence Manager. If the Transaction instance returned is not active, it cannot be used for transaction completion, but it can be used to set flags.
`newQuery`	The Persistence Manager instance is a factory for query instances, and queries are executed in the context of the Persistence Manager instance. The actual query execution might be performed by the Persistence Manager or might be delegated by the Persistence Manager to its database.

TABLE 5-3 `PersistenceManager` Methods *(Continued)*

Method	Description
`getExtent`	Returns a read-only Collection that contains all of the instances in the named class, and if the subclasses flag is true, all of the instances of the named class and its subclasses. The primary use for the collection returned as a result of this method is as a parameter to a Query instance. For this usage, the collection typically will not be instantiated in the JVM except if its elements are iterated. It is typically only used to identify the prospective database instances. You cannot call `PersistenceManager.getExtent` with the argument `subclasses=true`. The collection returned by `PersistenceManager.getExtent` may only be used within queries. The method `iterator` is the only supported method for an extent collection. Other collection methods—such as size and add—will throw either an `UnsupportedOperationException` or a `JDOUnsupportedOptionException`.
`getObjectById`	Returns a persistent instance that has the specified object identity in the cache. If no instance is active in the cache, it creates a hollow instance, populates its primary key fields with values from the ObjectId, and returns it. If the instance does not exist in the database, this method will not fail. But a subsequent access of the fields of the instance will throw an exception. Further, if a relationship is established to this instance, then the transaction in which the association was made will fail.
`getObjectId`	Returns the object identity of the specified instance. The identity is guaranteed to be unique only in the context of the Persistence Manager that created the identity, and only for the first two types of Identity—those that are managed by the application and those that are managed by the database (not supported for this release). Within a Persistence Manager instance, the ObjectId returned will be unique among all Instances associated with the Persistence Manager regardless of the type of ObjectId. If the application makes a change to the ObjectId returned by this method, there is no effect on the instance from which the ObjectId was obtained. That is, the returned ObjectId is a copy (clone) of local instance.
`getTransactionalInstance`	Returns a persistent instance valid for this instance of the Persistence Manager. Use this method when acquiring an instance for a Persistence Manager when the current instance is associated with a different Persistence Manager. `aPersistenceManager.getTransactionalInstance(pc)` is a shorthand for `aPersistenceManager.getObjectById(pc.getStateManager().getPersistenceManager().getObjectId(pc))`

TABLE 5-3 `PersistenceManager` Methods *(Continued)*

Method	Description
`makePersistent`	Inserts a persistent instance into the database. It must be called in the context of an active transaction. makePersistent will assign an object identity to the instance and transition it to persistent-new. During flush (using commit, or a user Query in a pessimistic transaction) of this instance, any transient instance reachable from this instance using persistent fields of this instance will behave as if the makePersistent method were executed on it, as well. This method throws JDOUserException if another object with the same ObjectIdentity is already associated with this Persistence Manager. This method has no effect on persistent instances managed by this Persistence Manager. It throws a JDOUserException if the instance is already managed by a different Persistence Manager.
`deletePersistent`	Deletes a persistent instance(s) from the database. It must be called in the context of an active transaction. The representation in the database will be deleted when this instance is flushed to the database (using commit, or user Query in pessimistic transaction). Note that this behavior is not exactly the inverse of makePersistent, due to the transitive nature of makePersistent. The implementation might delete dependent database objects depending on implementation-specific policy options (such as cascade delete). This method throws an exception if the instance is managed by a different Persistence Manager or if the instance is transient. This method has no effect on instances already deleted in the transaction.
`getPersistenceManagerFactory`	Returns the Persistence Manager Factory that created this Persistence Manager.
`setUserObject / getUserObject`	The application might manage persistent instances by using an associated object for bookkeeping. These methods let you manage the associated object. The parameter is not inspected or used in any way by the implementation.
`getProperties`	Transparent Persistence stores certain nonoperational properties and make those properties available to the application through a Properties instance. This method retrieves the Properties instance. Each key and value is a String. The keys required for this implementation are: • VendorName: The name of the vendor. • VersionNumber: The version number string.
`getObjectIdClass`	For the application to construct instances of the ObjectId class, there is a method that returns the ObjectId class given the persistence capable class.

TABLE 5-3 `PersistenceManager` Methods *(Continued)*

Method	Description
`newSCOInstance`	Returns a new Second Class Object instance of the type specified, with the owner and field name to notify upon changes to the value of any of its fields. If a collection class is created, then the class does not restrict the element types, allows nulls to be added as elements, and has an initial size of zero.
`newCollectionInstance`	Returns a new Collection instance of the type (or interface) specified, with the owner and field name to notify upon changes to the value of any of its fields. The collection class restricts the element types allowed to the elementType or instances assignable to the elementType, and allows nulls to be added as elements based on the setting of allowNulls. The Collection has an initial size as specified by the initialSize parameter.

Transactions

Insert and delete operations must occur within the context of a transaction. Transactions ensure the consistency of database reads and updates. They guard against system problems, such as disk crashes, that would corrupt the consistency of the database. Transactions also ensure that separate applications concurrently accessing and updating the same data within the database do so correctly. When you operate within the context of a transaction, it ensures that either all or none of your updates are written to the database.

Each Persistence Manager supports one transaction at a time, and this transaction applies to all of the transactional instances of persistence capable classes that it "owns." To work with the transaction, the application obtains the transaction object from the Persistence Manager:

```
Transaction myTrans = myPersistenceManager.currentTransaction();
```

In most cases the application will be running local transactions with a single database. The application starts and completes these transactions by calling transaction object methods:

```
Transaction txn=pm.currentTransaction();
txn.begin();
...operations...
txn.commit();
}
```

```
catch (Exception e) {
txn.rollback();
}
```

The Persistence Manager manages all interactions with the database, including refreshing cached copies of persistent data. The application needs only to identify transaction boundaries.

TABLE 5-4 discusses each method in detail.

TABLE 5-4 Transaction Methods

Method	Description
`begin`	Start a new Transaction. Throws a JDOUserException if the transaction is already active.
`commit`	The commit method performs the following operations: • Transitions a deleted instance to transient. • If retainValues is false, transitions persistent instances to the hollow state, clearing all non-primary-key fields. • If retainValues is true, transitions persistent instances to the persistent-nontransactional state, keeping all current field values.
`rollback`	The rollback method performs the following operations: • Transitions persistent-new instances to transient, restoring the fields to their pre-persistent values. • If retainValues is false, transitions persistent instances to the hollow state, clearing all non-primary-key fields. • If retainValues is true, transitions persistent instances to the persistent-nontransactional state, restoring the fields to their pre-modified values.
`getPersistenceManager`	Returns the Persistence Manager associated with this Transaction instance.
`isActive`	Tells whether there is an active transaction.
`getRetainValues` `setRetainValues`	If this flag is set to true, • `commit` transitions persistent instances to the persistent-nontransactional state, keeping all current field values. • `rollback` transitions persistent instances to the persistent-nontransactional state, restoring the fields to their pre-modified values. If this flag is set to false, • `commit` transitions persistent instances to the hollow state, clearing all non-primary-key fields. • `rollback` transitions persistent instances to the hollow state, clearing all non-primary-key fields.

TABLE 5-4 Transaction Methods *(Continued)*

Method	Description
`getOptimistic` `setOptimistic`	If this flag is set to true, then optimistic concurrency is used for managing transactions. The optimistic setting passed replaces the optimistic setting currently active. If set to true, then NontransactionalRead is set to true. The default is true.
`getNontransactionalRead` `setNontransactionalRead`	These methods access the flag that allows nontransactional instances to be read outside of a transaction. If this flag is set to true, then queries and navigation are allowed without an active transaction. If this flag is set to false, then queries and navigation outside an active transaction throw an exception. The default is true.
`getSynchronization` `setSynchronization`	Synchronization is supported for both managed and non managed environments. A Synchronization instance registered with the Transaction remains registered until changed explicitly by another setSynchronization. Only one Synchronization instance can be registered with the Transaction. If the application requires more than one instance to receive synchronization callbacks, then the application instance is responsible for managing them and forwarding callbacks to them. Any Synchronization instance already registered will be replaced. The beforeCompletion method will be called before the behavior specified for the transaction completion method commit. The `beforeCompletion` method will not be called before rollback. The `afterCompletion` method will be called after the transaction completion methods are finished. The parameter for the `afterCompletion (int status)` method will be either `Status.STATUS_COMMITTED` or `Status.STATUS_ROLLEDBACK`.

TABLE 5-4 Transaction Methods *(Continued)*

Method	Description

`QueryTimeout`
`UpdateTimeout`
	This method avoids deadlocks in the database by waiting a specified number of seconds before executing the query or update associated with this instance of the Transaction.
	The value is stored in seconds; zero means unlimited. For example:
	`tx.setQueryTimeout(6);`
	`tx.setUpdateTimeout(10);`
	PointBase does not currently support `PreparedStatement.setQueryTimeout()`. Add `,locks.timeout=value` to the URL or `pointbase.ini` file to use any other than default value (current default value is set to 60 seconds). However, be aware that `locks.timeout=0` sets the timeout to 0 seconds, rather than the `setQueryTimeout(0)` behavior of setting it to unlimited.
	`Locks.timout` is set on the server side, not the client side. This means the value will hold for all connections.

Transaction Isolation Levels

The transaction isolation level specifies the degree to which a transaction is separate from any concurrent transactions. Multiple users accessing the same database need to set a balance between performance and the degree of certainty in their view of the data. When accessing a database, certain inconsistencies can occur:

- Dirty read

  A read of uncommitted data. If Transaction A reads data from a database that has been modified by Transaction B, and the change is rolled back instead of being committed, Transaction A will have read data that is no longer correct.

- Nonrepeatable read

  Data returned by a query that would be different if the query were repeated within the same transaction. If one transaction reads a row, then another transaction updates or deletes the row and commits, the first transaction, on re-read, gets different data. Nonrepeatable reads can occur when other users are updating the same data you are reading.

- Phantom insert

  A read by one user that fetches a row that was inserted by another user's transaction. For example, one user's `SELECT` statement might select four rows from a table the first time it is executed and five rows the next item if a second user has, in the meantime, inserted a row that satisfies the first user's query.

Specifying a higher isolation level eliminates these inconsistencies, but decreases the performance of your application due to increased overhead, and leads to decreased system concurrency.

Transparent Persistence uses the default isolation level for the database (`TRANSACTION_READ_COMMITTED` for Oracle and MSSQL and `TRANSACTION_SERIALIZABLE` for PointBase).

`java.sql.Connection` uses the following SQL naming:

```
int TRANSACTION_NONE = 0;
int TRANSACTION_READ_UNCOMMITTED = 1;
int TRANSACTION_READ_COMMITTED   = 2;
int TRANSACTION_REPEATABLE_READ  = 4;
int TRANSACTION_SERIALIZABLE     = 8;
```

TABLE 5-5 shows which access inconsistencies are possible under each of these settings.

TABLE 5-5 Isolation Levels

Level	Dirty Read	Nonrepeatable Read	Phantom Insert
`TRANSACTION_READ_UNCOMMITTED`	Possible	Possible	Possible
`TRANSACTION_READ_COMMITTED`	Not Possible	Possible	Possible
`TRANSACTION_REPEATABLE_READ`	Not Possible	Not Possible	Possible
`TRANSACTION_SERIALIZABLE`	Not Possible	Not Possible	Not Possible

With the `TRANSACTION_NONE` setting, transactions are not supported at all.

Note – Oracle does not support `TRANSACTION_READ_UNCOMMITTED` or `TRANSACTION_REPEATABLE_READ`. Transparent Persistence does not validate any of the settings you use; unsupported settings will result in constraint violations from your database.

Concurrency Control

Programming in a database environment is transaction-based. Transactions ensure that multiple users concurrently accessing the database do so correctly—that is, transactions ensure the integrity of the database. This means that any insert or delete operations must be made within the context of a transaction.

Transparent Persistence handles concurrent transactions in two ways:

- Optimistic Transaction Management (default)

  With optimistic concurrency control, transactions assume that they will finish before another transaction changes the same data. The system assumes that the transaction will commit. However, it rolls back the transaction if it detects a conflict—that is, if another transaction changes the same data and commits while the first transaction is still in progress.

  When the application starts a transaction, the Persistence Manager records the beginning state of any database records it is using. Before committing the transaction, it compares the beginning state of the database records with the current state, to determine whether some other user has updated the database while the transaction was in progress.

- Data Store Transaction Management

  With data store transaction management, transactions are handled by the database and the specified transaction isolation level. See "Transaction Isolation Levels" on page 104 for more information.

  When the application starts a transaction, the Persistence Manager instructs the database itself to begin a transaction. This means that between the first data access until the commit, there is an active database transaction.

The Persistence Manager Factory has methods that let you set the default concurrency management strategy. The Transaction object has methods that let you set the concurrency management strategy before beginning a transaction.

Optimistic transactions take longer to execute than database transactions. This is because each optimistic transaction consists of two database transactions: one read transaction for the query, which is closed at query completion, and a write transaction for the commit. Additionally, the transaction to commit the updates is labor-intensive for the database, because it must check for rows that match the originally selected object. However, optimistic transactions allow for optimal concurrency, because database records are locked for a minimal amount of time.

If an optimistic transaction fails, you receive an exception with an attached failed object array.

Recovery of database transactions is handled by the database. For example, the database may check for deadlocks or timeouts, and then cancel or roll back the transaction appropriately.

You should use optimistic transactions when you will have transactions that involve user "think time," such as within web applications. Use database transactions when you will have transactions that are executed quickly on a server (for example, in batch applications or within the method of a stateless session bean or a servlet).

Retain Values

You can set the Persistence Manager to retain values outside of the context of a transaction. This is most beneficial for optimistic transactions or for selecting data outside the context of any transactions. This means that data is cached locally, even outside the context of a transaction. This allows faster access of the data, but you might risk having stale data in your local cache if the database was updated outside of the IDE.

If you turn off `retainValues`, then the fields in the default fetch group are reread the first time one of them is accessed. Each field not in the default fetch group is read in once when it is first accessed.

Note – If `retainValues()` is set to true, the following situations occur:

```
tx.begin();
Object o1 = c.get(i);
c.add(o);      // This will cause reload, and will remove all
               // existing duplicate elements.
o1 == c.get(i); // This can return true or false depending on the
                     // contents of the new collection.
```

Coding With Optimistic Concurrency Control

Setting the `Optimistic` flag to `true` has the side effect of setting the `NontransactionalRead` flag to `true` as well.

With optimistic concurrency control, the less time your transactions are open, the more likely they are to commit successfully. The longer a transaction is open, the greater the risk of another transaction modifying data that is involved in your transaction. If the system detects that another transaction has modified data that you are trying to change, it throws a `JDODataStoreException` during flush or commit, and you will need to roll back the transaction.

Optimistic transactions are useful when there are long-running transactions that rarely affect the same instances. In these cases, the database will exhibit better performance by deferring database exclusion on modified instances until commit.

With optimistic transactions, instances queried or read from the database will not be transactional unless they are modified, deleted, or marked by the application as transactional in the transaction.

At commit time, instances that have been made transactional will be verified against the current contents of the database, to ensure that the state in the database is the same as the “before image” of the instance in the transaction.

If any instance is found to have changed, an exception is thrown that contains the list of instances that failed the verification. The optimistic transaction stays active, and you need to roll back the transaction.

In the case of concurrent updates, Transparent Persistence applications running in optimistic mode throw a `JDODataStoreException`.

Optimistic transaction management is specified by the `Optimistic` setting on `Transaction`.

At flush or commit, only fields in the same fetch group are checked for concurrent changes.

When you are ready to actually commit your data modifications to the database, the system checks if that data has been changed by any other transaction since the time your transaction first read the data. If the data has not been changed, then your transaction can complete. If any data has been changed, then you need to roll back your updates.

Note – When Transparent Persistence rolls back a transaction because of a concurrency conflict, it is likely that one or more of the original values have been changed by another transaction.

Coding With Data Store Concurrency Control

The data store concurrency control approach depends on the particular database you are using, and how you have set the isolation level.

Under the data store approach, after you update an object, you can proceed with your transaction and be assured of a successful commit, unless a deadlock or error occurs.

Deadlocks occur in situations where multiple transactions attempt to update the same sets of records. For example, one transaction locks record `A` and waits to obtain a lock on record `B`. At the same time, another transaction has locked record `B` and is waiting to obtain a lock on record `A`. Neither transaction relinquishes the lock it already holds, and they both deadlock because they are waiting for locks that they will never acquire. Different database management systems handle deadlock situations differently.

For example, in an application using Transparent Persistence, transaction `A` successfully updates persistent object `01`, and then tries to update persistent object `02`. Concurrently, another transaction, `B`, successfully updates persistent object `02`, and then tries to update persistent object `01`, causing a deadlock in the database. You

might get a deadlock even if one transaction had read `O1` and wanted to update `O2`, and the other transaction had read `O2` and wanted to update `O1`. The outcome of this deadlock depends on which DBMS you are using.

Microsoft SQL Server does not detect deadlocks. You need to call `setQueryTimeout()` and `setUpdateTimeout()` on the transaction to specify the amount of time the query should wait before timing out. The default is to wait forever.

In contrast, Oracle detects deadlocks between concurrent transactions only when one user commits a conflicting transaction. In such situations, the first committed transaction succeeds; the other transaction is rolled back.

In general, keep data store concurrency transactions short to avoid locking out other transactions. Lockouts are less of a problem if you are dealing with applications that run under exclusive control—that is, applications that gain control over a portion (or all) of a database and exclude all other applications, such as an accounts payable check-generating application.

Accessing the Database

This section specifies the life cycle for persistence-capable class instances. The classes include behavior as specified by the class (bean) developer and additional behavior as provided by the reference enhancer or Transparent Persistence. The enhancement of the classes allows application developers to treat Transparent Persistence instances as if they were normal instances, with automatic fetching of the persistent state from the database.

A persistence-capable class has persistent fields and relationship fields that model a class of data in a database. For an application to actually work with specific entities from the database, it must create and work with instances of the persistence-capable class that models the data. If, for example, the application is using an `Employee` class that models the employee database table, the application needs instances of that `Employee` class.

After the application has persistent instances that represent data, the behavior of each instance is linked to the transactional store with which it is associated. Transparent Persistence automatically tracks changes made to the values in the instance, and automatically refreshes values from the database and saves values into the database as required to preserve the transactional integrity of the data. This means that application code can operate on the persistent instances as Java instances, and the Transparent Persistence runtime environment will perform all of the database interactions indicated by the application's actions.

During the life of a persistent instance, it transitions among various states until it is finally garbage collected by the JVM. During its life, the state transitions are governed by the behaviors executed on it directly as well as behaviors executed on the Persistence Manager by both the application and by the execution environment.

During the life cycle, instances at times might be inconsistent with the database as of the beginning of the transaction. If instances are inconsistent, they are called “dirty”. Instances made newly persistent, deleted, or modified in the transaction are dirty.

At times, Transparent Persistence stores the state of persistent instances in the database. This process is called “flushing,” and it does not affect the dirty state of the instances.

This section summarizes the ways in which applications can create and work with instances of persistence-capable classes. It also introduces some of the terminology Transparent Persistence uses for instance manipulation and instance status. Instance status is primarily maintained for the runtime environment, but the application might occasionally need to check it or reset it.

Overflow Protection

Write protection for the database is handled by the database driver. Transparent Persistence does not do any separate write validation.

When reading from the database, you will get a JDOUserException if the value returned from the database is a number less than the `MIN_VALUE` or greater than the `MAX_VALUE` allowed for the field type. For example, you might set the values of `short` to be between -32768 and 32768, inclusive:

```
java.lang.Short:
public static final short MIN_VALUE = -32768;
public static final short MAX_VALUE = 32767;
```

The overflow validation on read is done for types `short`, `int`, `long`, `byte`, `Short`, `Integer`, `Long`, and `Byte`.

Inserting Persistent Data

When the client supplies data for a new record, the application handles it by creating a new persistent instance:

```
Employee newEmployee = new Employee(<data>);
// Instance status is now "transient."
pMgr.makePersistent(newEmployee);
// Instance status is now "persistent-new."
```

When the transaction is committed, the Transparent Persistence runtime environment generates an SQL insert operation (or its equivalent) for the data encapsulated in this instance.

This is a two-step process. When the `newEmployee` instance is constructed, it is not associated with the persistence manger and is not automatically saved when the transaction ends. The `makePersistent()` call associates the `newEmployee` instance with the Persistence Manager, which manages its values for the application.

Updating Persistent Data

When an application needs to change data in a persistent instance it does so by acting directly on the instance:

```
selectedEmployee.setVacationHours(132);
// Instance status is now "dirty."
```

When the transaction is committed, the Transparent Persistence runtime environment generates an SQL update operation (or its equivalent) for the data encapsulated in this instance. After the transaction commits, the instance's status will be reset.

Transparent Persistence does not support updates to SCO Collections that cause the removal of an element by index, because the underlying collection can be changed during the update operation by way of a refetch from the database.

Deleting Persistent Data

When the application needs to delete data represented by a persistent instance, it does so by calling a Persistence Manager method:

```
persistenceManager.deletePersistent(selectedEmployee);
// Instance status is now marked for deletion.
```

When the transaction is committed, the Transparent Persistence runtime environment generates SQL delete operation (or its equivalent) for the data represented by this instance.

Transparent Persistence supports two types of delete semantics:

- None (default)

  If an object is deleted, related objects in one-way relationships are left untouched.

  In a managed relationship, the relationships between the deleted object and related objects are nullified.

- Cascade

  If an object is deleted, all related objects are deleted at flush or commit.

  For example, consider the classes `Department` and `Employee`, where `Department` has an `Employee` Collection, and `Employee` has a reference to a `Department`.

  If the `Employee` relationship is marked for cascade delete, deleting a `Department` instance will also delete all `Employee` instances associated with this `Department`.

  If the `Department` relationship is marked for cascade delete, deleting an `Employee` instance will also delete the `Department` instance referenced from this `Employee`. It will not delete other `Employee` instances associated with that `Department` unless `Employee` relationships are marked for cascade delete as well.

You can specify the deletion method in the Delete Action field of the Properties for a persistence-capable class. See "Setting Options and Properties" on page 71.

Note – Setting cascade delete on the many side of a one-to-many or many-to-many relationship can result in unwanted deletions. Cascade delete should be set only on one-to-one relationships or on the one side of a one-to-many relationship.

An example of deleting all objects on one side of a many-to-many relationship would be deleting all projects from a relationship between projects and employees. The code would be as follows:

```
Collection p = e.getProjects();
Object[] a = p.toArray();
p.clear();
pm.deletePersistent(a);
```

Querying the Database

Queries allow you to access persistent data without writing separate SQL statements. You can run your code on any of a number of different databases, and you can re-map the persistence-capable classes to a different database, possibly with a different schema, without changing the code.

When the application needs data from the database, it uses the `newQuery()` method to obtain a Query object from the Persistence Manager, uses methods from the Query interface to define a query, and executes the query. The following example shows how this is done:

```
Class empClass = Employee.class;
Collection empExtent = pMgr.getExtent(empClass, false);
String empFilter = "id == 59439";
Query q = pMgr.newQuery(empClass, empExtent, empFilter);
Collection result = (Collection) q.execute();
```

A query is defined by the elements shown in TABLE 5-6.

TABLE 5-6 Query Elements

Element	Requirement	Description
Candidate class	Required	This defines the class of the instances in the candidate collection that are considered for this query. The class is used to scope the names in the query filter. The candidate class of a query must be persistence-capable. It is defined by a `newQuery` argument or by the Query method `setClass`.
Candidate collection	Required	This is the extent collection (see the `PersistenceManager.getExtent` method) of the candidate class and defines the input collection for the query. It is defined by a `newQuery` argument or by the Query method `setCandidates`. Querying memory collections is not supported; the extent collection is the only valid candidate collection for a query.
Query filter	Required	The filter is a String that specifies which objects from the candidate collection are returned by the query. It is defined by a `newQuery` argument or by the Query method `setFilter`. The default is "true", which means that all instances are returned.
Query parameters	Optional	A query might have one or more parameters that are bound to actual values at query execution time. The definition follows the syntax for formal parameters in the Java language. It is defined by the Query method `declareParameters`.
Query variables	Optional	The query filter might use unbound variables in order to navigate a collection relationship. It follows the syntax for local variables in the Java language. It is defined by the Query method `declareVariables`.
Import statements	Optional	Parameters and variables might come from a class other than the candidate class, and the names might need to be declared in an `import` statement to eliminate ambiguity. The syntax is the same as in the Java `import` statement. It is defined by the Query method `declareImports`.
Ordering	Optional	You can order the result set by a field of the candidate class. The ordering specification includes the list of fields with the ascending/descending indicator. It is defined by the Query method `setOrdering`.

The Persistence Manager is the factory of Query instances and queries are executed in the context of a Persistence Manager. Any persistence-capable instances returned by the query are associated with the Persistence Manager and its transaction. This Persistence Manager's automatic update/refresh process will include these instances. There might be multiple query instances active in the same Persistence Manager.

Use a `newQuery()` method in the Persistence Manager for each query you want to create. The preceding example constructs a query instance with the candidate class, candidate collection, and filter specified. Other options are shown in TABLE 5-7.

TABLE 5-7 `newQuery` Options

Method	Description
`Query newQuery()`	Construct an empty query instance.
`Query newQuery (Object query)`	Construct a query instance from another query. The parameter might be a serialized/restored Query instance from a different execution environment, or the parameter might be currently bound to a Persistence Manager. Any of the elements Class, Filter, Import declarations, Variable declarations, Parameter declarations, or Ordering from the parameter Query are copied to the new Query instance, but a candidate collection element is discarded.
`Query newQuery (Class cls)`	Construct a query instance with the candidate class specified.
`Query newQuery (Class cls, Collection cln)`	Construct a query instance with the candidate class and candidate collection specified.
`Query newQuery (Class cls, String filter)`	Construct a query instance with the candidate class and filter specified.
`Query newQuery (Class cls, Collection cln, String filter)`	Construct a query instance with the candidate class, the candidate collection, and filter specified.

TABLE 5-8 discusses each method of the `Query` interface in detail.

TABLE 5-8 Query Interface Methods

Method	Description
`void setClass (Class resultClass)`	Binds the candidate class to the query instance.
`void setCandidates (Collection candidateCollection)`	Binds the candidate collection to the query instance.
`void setFilter (String filter)`	Binds the query filter to the query instance.
`void declareParameters (String parameters)`	Binds the parameter declarationsto the query instance. This method defines the parameter types and names that will be used by a subsequent execute method.
`void declareVariables (String variables)`	Binds the unbound variable declarations to the query instance. This method defines the types and names of variables that will be used in the filter but not provided as values by the execute method.
`void declareImports (String imports)`	Binds the import statements to the query instance.
`void setOrdering (String ordering)`	Binds the ordering statements to the query instance.
`void setIgnoreCache (boolean flag); boolean getIgnoreCache ()`	Allows you to request that queries be optimized to return approximate results by ignoring changed values in the cache. This option is only useful for optimistic transactions and allows the database to return results that do not take modified cached instances into account. `setIgnoreCache (false)` is not supported.
`void compile ()`	Requires the Query instance to validate any elements bound to the query instance and report any inconsistencies by throwing an exception.

The `Query` interface provides methods that execute the query based on the parameters given. `Query.execute` always returns a collection of objects. In the preceding example, the query selects a single object, but the dynamic type of the result of `q.execute` is `Collection`. This means that you must iterate through the result collection and `Iterator.next` returns the `Employee`.

Query Filters

The query filter is a Java Boolean expression that is evaluated for each instance in the collection. If no filter is specified, the default is `true`, which filters the input collection only for class type.

Simple Filter Expressions

The simplest form is a relational expression that compares a candidate class field with a literal value:

```
q.setFilter("id == 59439");
```

You can also include the Boolean operators `&`, `&&`, `|`, `||` and `!` as well as the arithmetic operators `+`, `-`, `*`, and `/`. For example, in the following code, the first line filters elements with a first name of John and a last name of Jones. The second line filters elements with a first name of John or a salary greater than 200,000.

```
q.setFilter("firstname == \"John\" & lastname == \"Jones\"");
q.setFilter("firstname == \”John\” | salary > 200000.0");
```

Identifiers in the filter expression denote fields of the candidate class, unless the name is defined as a parameter, variable, or imported as a class name. For example, `firstname`, `lastname`, and `salary` are fields of the `Employee` class. As in the Java language, `this` is a reserved word referring to the element of the candidate collection being evaluated.

The following filter expressions are equivalent:

```
q.setFilter("firstname == \"John\"");
q.setFilter("this.firstname == \"John\"");
```

Any assignment, pre- and post-increment, and pre- and post-decrement operators are not allowed. Therefore, filter expressions do not have a side effect on the objects to be returned. The supported method calls are `Collection.contains`, Collection.isEmpty, `String.startsWith` and `String.endsWith`. In contrast to the Java language, equality and ordering comparisons between primitives and instances of wrapper classes are valid, as are equality and ordering comparisons of `Date` fields and `Date` parameters. You can also include other relational operators `<`, `<=`, `>`, `>=` and `!=`, and Boolean operators such as `&` and `&&`.

Query Parameters

A query parameter is the only part of a query definition that is not fixed at query declaration. A parameter's actual value is passed to the `execute` method. The following query returns the employees with a first name specified by the `execute` method call:

```
Class empClass = Employee.class;
String filter = "firstname == name";
Collection empExtent = pMgr.getExtent(empClass, false);
String param = "String name";
Query q = pMgr.newQuery(empClass, empExtent, filter);
q.declareParameters(param);
Collection result = (Collection) q.execute("John");
```

Here `firstname` denotes a field in the persistence-capable class `Employee`, and `name` denotes the query parameter name. The actual value of the parameter name is specified as an argument of `execute`. The call `q.execute("John")` returns a collection of `Employee` instances with a `firstname` value of `John`. You can reuse the same query instance to return `Employee` instances with a different name by calling `execute` again and passing a different parameter value, as in `q.execute("Sue")`.

The declaration of the query parameter defines the name and type of the query parameter. The actual value passed to `execute` must be compatible with the parameter type. A query can define multiple parameters. The parameters passed to `execute` associate in order with the parameter declarations.

Each parameter of the `execute` method is an object that is either the value of the corresponding parameter or the wrapped value of a primitive parameter.

Note – Any parameters passed to the `execute` methods are used only for the current execution, and are not remembered for future execution.

The methods from the query API that define query elements `setClass`, `setCandidates`, `setFilter`, `declareImports`, `declareParameters`, `declareVariables`, and `setOrdering` are replacing, not additive. This means if

these methods are called twice before query execution, the second call overwrites the settings from the first call. In the following sample code, the query is defined taking a single parameter called `lastname`:

```
Query query = pm.newQuery(Employee.class);
query.declareParameters("String firstname");
query.declareParameters("String lastname");
...
```

If you want to create a query taking two parameters, you have to define them in a single `declareParameters` call:

```
Query query = pm.newQuery(Employee.class);
query.declareParameters("String firstname, String lastname");
```

Relationship Navigation

The query filter may navigate a relationship the same as in the Java language. The following query returns `Employee` instances where the value of the `name` field in the associated `Department` instance is equal to the value passed as a parameter:

```
Class empClass = Employee.class;
String filter = "department.name == depName";
Collection empExtent = pm.getExtent (empClass, false);
String param = "String depName";
Query q = pm.newQuery (empClass, empExtent, filter);
q.declareParameters (param);
Collection emps = (Collection) q.execute ("R&D");
```

Query variables are used to navigate a collection relationship. The filter expression includes a call of the method `Collection.contains` to specify the scope of the variable. The call is followed by a Boolean expression that defines the condition for the instances in the collection relationship. The following query selects all `Department` instances containing at least one `Employee` instance with a salary

greater than the value passed as a parameter. The expression `emps.contains (emp)` defines the `Employee` collection relationship as the scope of the variable `emp`, and `emp.salary > sal` defines the condition for the `Employee` instances.

```
Class depClass = Department.class;
Collection deptExtent = pm.getExtent (depClass, false);
String imports = "import mypackage.Employee";
String vars = "Employee emp";
String filter = "emps.contains (emp) & emp.salary > sal";
Query q = pm.newQuery (depClass, deptExtent, filter);
q.declareParameters (param);
q.declareVariables (vars);
Collection deps = (Collection) q.execute (new Float (30000.));
```

Transparent Persistence supports comparing relationships fields with persistent instances. For example, a filter expression could compare the `department` field of `Employee` with a `Department` query parameter: `"department == dept"`.

Note – Transparent Persistence does not support multiple `contains` clauses for the same variable. A declared variable must be used in a filter.

Ordering Specification

The following query selects all `Employee` instances having a salary greater than `30000`, in ascending order of salary:

```
Class empClass = Employee.class;
Collection empExtent = pMgr.getExtent(empClass, false);
String empFilter = "salary > 30000.0";
Query q = pMgr.newQuery(empClass, empExtent, empFilter);
q.setOrdering("salary ascending");
Collection result = (Collection) q.execute();
```

The parameter passed to `setOrdering` allows multiple ordering declarations separated by commas. The result set is ordered using the first ordering expression. Those entries where the first ordering expression yields the same value are ordered using the second ordering expression, then the third ordering expression, and so on. You can specify an ordering expression that includes relationship navigation as well.

The following ordering declaration causes the query above to return Employees in ascending order of the name of the associated department. Employees from the same department are ordered by salary.

```
q.setOrdering(“department.name ascending, salary ascending”);
```

String Operations

String fields and values in filter expression are compared using the `==` and `!=` operators. Transparent Persistence supports wild card queries using the String methods `startsWith` and `endsWith`. The following filter expression selects all `Employee` instances having a first name that starts with `M`:

```
String empFilter = "firstname.startsWith("M");
```

Queries in Optimistic and Data Store Transactions

A query executed in a data store transaction first flushes changes from the transaction and then evaluates the query in the data store. This means the query result reflects any changes made in this transaction prior to query execution. In optimistic transactions, there is no flushing, so the query result might not reflect current changes or might include instances that do not satisfy the query result because of recent changes in the transaction. You can execute a query outside of a transaction if nontransactional reads are allowed.

Expression Capabilities

Following are the capabilities of the expressions supported by Transparent Persistence:

- Operators applied to all types where they are defined in the Java language, as shown in TABLE 5-9:

TABLE 5-9 Query Operators

Operator	Description
==	equal
!=	not equal
>	greater than

TABLE 5-9 Query Operators *(Continued)*

Operator	Description
<	less than
>=	greater than or equal
<=	less than or equal
&	Boolean logical AND (not bitwise)
&&	conditional AND
|	Boolean logical OR (not bitwise)
||	conditional OR
~	Boolean or integer bitwise invert
+	binary or unary addition or String concatenation
-	binary subtraction or numeric sign inversion
*	times
/	divide by
!	logical invert

- Parentheses to explicitly mark operator precedence
- Cast operator (class)
- Promotion of numeric operands for comparisons
- Equality and ordering comparison and arithmetic operations on object-valued fields of wrapper types (`Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`) and of `BigDecimal` and `BigInteger`

  This uses the wrapped values as comparands or operands.
- Equality comparison of object-valued fields of `PersistenceCapable` types

  This uses the Transparent Persistence Identity comparison of the references. Thus, two objects will compare equal if they have the same Transparent Persistence Identity.
- Equality comparison of object-valued fields of `non-PersistenceCapable` types

  This uses the `equals` method of the field type.
- String concatenation

  Only concatenation of strings is supported. For example, `String + primitive` is not supported.

Examples

This section includes several examples of typical queries. Each example is accompanied by a description and its equivalent ANSI SQL statement.

The examples use the following definitions for persistence-capable classes:

```
package com.xyz.hr;
class Employee {
String name;
Float salary;
Department dept;
Employee boss;
}
package com.xyz.hr;
class Department {
String name;
Collection emps;
}
```

Single-Table Select

This query selects all Employee instances from the extent.

```
ANSI SQL equivalent: SELECT * FROM EMPLOYEE

Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "true";
Query q = pm.newQuery (empClass, clnEmployee, filter);
Collection emps = (Collection) q.execute ();
```

Single-Table Select With Constraint

This query selects all Employee instances that have a field value that passes a Boolean test; in this case, where the salary is greater than the constant 30000.

```
ANSI SQL equivalent: SELECT * FROM EMPLOYEE WHERE SALARY > 30000
```

The `Float` value for `salary` is unwrapped for the comparison with the literal value. If the value for the `salary` field in the candidate instance is `null`, it cannot be unwrapped for the comparison, and the candidate instance is rejected.

```
Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > 30000.00";
Query q = pm.newQuery (empClass, clnEmployee, filter);
Collection emps = (Collection) q.execute ();
```

Single-Table Select With Parameterized Constraint

This query selects all Employee instances that have a field value that passes a Boolean test that uses a parameter; in this case, where the salary is greater than the value passed as a parameter.

```
ANSI SQL equivalent: SELECT * FROM EMPLOYEE WHERE SALARY > ?
```

The parameter declaration is a `String` containing one or more parameter type declarations separated by commas. This follows the Java syntax for method signatures.

If the value for the `salary` field in a candidate instance is `null`, then it cannot be unwrapped for the comparison, and the candidate instance is rejected.

```
Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "salary > sal";
String param = "Float sal";
Query q = pm.newQuery (empClass, clnEmployee, filter);
q.declareParameters (param);
Collection emps = (Collection) q.execute (new Float (30000.));
```

Single-Table Select With Ordering Clause

This query selects a list of objects ordered by the value of one or more of the object's fields.

The ordering statement is a `String` containing one or more ordering declarations separated by commas. Each ordering declaration is the name of the field in the name scope of the target class followed by `ascending` or `descending`.

```
ANSI SQL equivalent: SELECT * FROM EMPLOYEE ORDER BY LASTNAME
ASCENDING, FIRSTNAME ASCENDING

Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "true";
Query q = pm.newQuery (empClass, clnEmployee, filter);
query.setOrdering("lastname ascending, firstname ascending")
Collection emps = q.execute ();
```

Join Across a "to-one" Relationship

This query selects a list of objects that have a referenced object that matches a Boolean test; in this case, where the value of the name field in the Department instance associated with the Employee instance is equal to the value passed as a parameter.

```
ANSI SQL equivalent: SELECT EMPLOYEE.* FROM EMPLOYEE, DEPARTMENT
WHERE DEPARTMENT.DEPTNAME = ? AND EMPLOYEE.DEPTID =
DEPARTMENT.DEPTID
```

If the value for the `dept` field in a candidate instance is `null`, then it cannot be navigated for the comparison, and the candidate instance is rejected.

```
Class empClass = Class.forName("com.xyz.hr.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "dept.name == name";
String param = "String Engineering";
Query q = pm.newQuery (empClass, clnEmployee, filter);
q.declareParameters (“String name”);
Collection emps = (Collection) q.execute (“Engineering”);
```

Join Across a "to-many" Relationship

This query selects a list of objects that have one or more objects in a referenced collection that match a Boolean test; in this case, all Department instances where the collection of Employee instances contains at least one Employee instance having a salary greater than the value passed as a parameter.

```
ANSI SQL equivalent: SELECT DEPARTMENT.* FROM DEPARTMENT, EMPLOYEE
WHERE EMPLOYEE.SALARY > 30000 AND DEPARTMENT.DEPTID =
EMPLOYEE.DEPTID

Class depClass = Class.forName("com.sun.xyz.Department");
Collection clnDepartment = pm.getExtent (depClass, false);
String vars = "Employee emp";
String filter = "emps.contains (emp) & emp.salary > sal";
String param = "float sal";
Query q = pm.newQuery (depClass, clnDepartment, filter);
q.declareParameters (param);
q.declareVariables (vars);
Collection deps = (Collection) q.execute (new Float (30000.));
```

Overlapping Primary Key and Foreign Key

Transparent Persistence supports overlapping primary and foreign keys, but there are several issues to be aware of. As an example, consider the following schema:

```
CREATE TABLE Order
(
        orderNumber INT PRIMARY KEY,
        customerName VARCHAR2(32) NULL,
        requestedDate DATE NULL
)
CREATE TABLE LineItem
(
        lineItemNumber INT NOT NULL,
        orderNumber INT NOT NULL,
        price FLOAT NOT NULL,
        description VARCHAR2(100) NULL,
        PRIMARY KEY (lineItemNumber, orderNumber),
        FOREIGN KEY (orderNumber) REFERENCES Order(orderNumber)
)
```

The persistence-capable classes would look as follows:

```
public class Order
{
        int ordernumber;
        String customername;
        Date requesteddate;
        HashSet lineitems;
}
public class Lineitem
{
        int lineitemnumber;
        int ordernumber;
        float price;
        String description;
        Order order;
}
```

Since Transparent Persistence does not support modifying primary keys, it does not support modifying the relationship between `Order` and `Lineitem`. For example, in order to add a `Lineitem` to an `Order`, you would need to modify the `Lineitem.ordernumber`, which is part of the primary key. Similarly, if you try to remove a `Lineitem` from an `Order`, you would need to set the `Lineitem.ordernumber` to zero, which could cause a constraint violation in the database. In both cases, Transparent Persistence would not update the Oids nor rehash the instances in the cache.

To deal with this situation, use the guidelines in the following sections:

Creating an `Order/Lineitem` Relationship

For this example, the code below creates an `Order/Lineitem` relationship:

```
tx.begin();
Order o = new Order();
o.setOrdernumber(1);
o.setCustomername("peter");
HashSet items = new HashSet();
o.setLineitems(ltems);
Lineitem lt = new Lineitem();
lt.setLineitemnumber(1);
lt.setOrdernumber(1);
```

You need to explicitly set the `ordernumber` to the `ordernumber` of an existing `Order`. The `Order` can either be persistent in the database already or it can be in the process of being made persistent.

```
items.add(lt);
```

Note – Once the `Lineitem.ordernumber` is set to 1, it can only be added to Order 1's `lineitems` collection.

```
pm.makePersistent(o);
tx.commit();
```

Deleting `Order/Lineitem` Relationship

The code example below properly removes an `Order/Lineitem` relationship.

```
tx.begin();
Order o = ....            // fetch the Order
Lineitem lt = .....       // get the Lineitem you want to remove
```

You can remove a `Lineitem` from an `Order` as long as you explicitly delete it within the same transaction. Note that you can interchange the following two lines

```
pm.deletePersistent(lt);
o.getLineitems().remove(lt);
```

Similarly, you can remove all `Lineitems` from an `Order` as long as you explicitly delete them all within the same transaction.

```
pm.deletePersistent(o.getLineitems());
o.getLineitems().clear();

tx.commit();
```

Restrictions

Following is the list of restrictions:

- Moving a `Lineitem` from one `Order` to another is not supported. You need to remove or delete it from one `Order` and create a new one to be added to another `Order`.
- `Lineitem.setOrder()` is not supported. For example:

```
Lineitem lt = o.getLineitems().get(0);
```

This will throw a `JDOUnsupportedOptionException`.

```
lt.setOrder(null);
```

The following lines of code will cause a `JDOUserException` at commit time:

```
o.getLineitems.add(lt);
lt.setOrder(o);
```

Fetch Groups

A fetch group is a group of persistent fields that will be retrieved together. When an application requests the value of one field in the group, values for all fields in the group are loaded together. This provides more efficient transfer of values that are frequently used together, such as the fields that make up an employee address. The class developer can analyze the fields in the database record and decide whether adding fetch groups to the class definition will improve performance of the class.

You can specify Level, Independent, Default, or None. There are two types of settings, hierarchical and independent.

Hierarchical groups include the Default and Level settings, and build on one another. A setting of Default for a field means that field will be fetched along with all other fields that have a setting of Default. When a field in the Level 1 group is fetched, all fields in group Level 1 and the Default group are fetched as well.

By default, Transparent Persistence includes all persistent fields except relationship fields in the Default fetch group. If the Fetch Group property is disabled, the field is not persistent, not mapped, or is a key field and will always be fetched. Relationship fields must have a setting of None.

Checking Instance Status

The preceding discussions of basic operations queries, updates, and so on, have touched on the status of persistent instances and demonstrated some of the ways in which the Persistence Manager sets the status of instances it is managing, and then uses that status to determine which operations are required at transaction boundaries. If necessary the developer can check and reset the status of instances.

The recommended approach for applications to interrogate the state of the instance is to use the class `JDOHelper`. This class provides static methods that delegate to the instance if it implements `PersistenceCapable`, and if not, returns the values that would have been returned by a transient instance.

Methods available include, but are not limited to, the following:

```
isDirty()
makeDirty()
```

Transparent Persistence Identity

Java defines two concepts for determining whether two instances are the same instance or whether they represent the same data:

- Java object identity is entirely managed by the JVM. Instances are identical if and only if they occupy the same storage location within the JVM.
- Java object equality is determined by the class. Instances are equal if they represent the same data, such as the same value for an integer or equivalent bits in a bit array.

The interaction between Java object identity and equality is important for Transparent Persistence developers. Java object equality is application-specific, and Transparent Persistence does not change the application's implementation of equality. There is only one instance in each Persistence Manager representing the persistent state of each corresponding database object. Therefore, Transparent Persistence defines object identity differently from both the JVM object identity and the application equality.

Applications should implement equality for persistence-capable classes differently from the default implementation, which uses the JVM object identity. This is because the JVM object identity of a persistent instance cannot be guaranteed between Persistence Managers and across space and time, except in very specific cases.

If persistent instances are stored in the database and are queried using the == query operator or are referred by a persistent collection that enforces identity (Set, Map), then the implementation of equals should exactly match the Transparent Persistence implementation of equality, using the primary key or Oid as the key. This is not enforced, but if not correctly implemented, the semantics of collections can differ.

To avoid confusion with Java object identity, this manual refers to the Transparent Persistence concept as Transparent Persistence identity. Transparent Persistence identity is used for databases in which the values in the instance determine the identity of the object in the database. Transparent Persistence identity is managed by the application and enforced by the database.

The Persistence Manager manages instance identity for the developer, but then when comparing persistent instances (for example, with the = = operator), it is the Transparent Persistence Oids that are compared.

Oid Class

The Oid class (Object ID) is specific for each persistence-capable class. It is a characteristic of the persistence-capable class and must be created at mapping time.

Each Persistence Manager must manage the cache of Transparent Persistence instances so that only one such instance is associated with each Persistence Manager that encapsulates a database object.

To accomplish this, each Transparent Persistence class has an associated Oid class that includes a field or fields whose values uniquely identify a Transparent Persistence instance. Each instance of a Transparent Persistence class has an associated instance of the ID class that holds the identifier. This allows the runtime environment to compare Oids and manage identity and equality of the Transparent Persistence instances.

With many databases, the identity of an entity is determined by a value found in the data. This is typical of relational database systems, in which each row or object has a key value that identifies it. For this kind of database, the Oid class created by the Java generator is a “primary key class,” with a field that holds the primary key value.

An Oid class can be either of the following types:

- Static nested class with the suffix `Oid` (default)
- Separate class with suffix `Key`

Both suffixes are case-insensitive.

As an example, of the name of the persistence-capable class is `mypackage.Employee`, valid Oid class names are `mypackage.Employee.Oid` or `mypackage.EmployeeKey`.

Note – For each field of a persistence-capable class, the Properties window has a Boolean Key Field option. However, the Oid class defines key fields as those fields in the persistence-capable class that have matching (public) fields in the Oid class of equal name and type.

To avoid a conflict, you need to ensure that the Key Field settings of your persistence-capable classes match the structure of your Oid classes:

- A field in the persistence-capable class marked as a primary key must be declared in the Oid class
- A field in the Oid class must be marked as a primary key and be present in the persistence-capable class
- A persistence-capable class and an Oid class field of same name must be of consistent types

Uniquing

Transparent Persistence identity of persistent instances is managed by the implementation. For a managed Transparent Persistence identity, only one persistent instance is associated with a specific database object per Persistence Manager instance, regardless of how the persistent instance is acquired:

- `PersistenceManager.getObjectById(Object oid)`
- Query via a `Query` instance associated with the Persistence Manager instance
- Navigation from a persistent instance associated with the Persistence Manager instance
- `PersistenceManager.makePersistent(Object pc)`
- `PersistenceManager.getTransactionalInstance(Object pc)`

A primary key identity is associated with a specific set of fields. The fields associated with the primary key are a property of the persistence-capable class and cannot be changed after the class is enhanced for use at runtime. When a transient instance is made persistent, the implementation uses the values of the fields associated with the primary key to construct the Transparent Persistence identity.

Mapping

For each persistence-capable class, the Java Generator generates a public static nested class called `Oid`. You can access this class with `<className>.Oid`. At the time of generation, you specify whether a class is persistence-capable. The GUI does not protect the primary key fields of each persistence-capable class and the fields of

<classname>.Oid from changes You must maintain consistency between the names and types of primary key fields of persistence-capable classes and the names and types of fields of <classname>.Oid.

The following example creates and accesses an Oid class for class Employee:

```
Employee.Oid eieio = new Employee.Oid();
eieio.id = 142857;
Employee emp = (Employee) myPM.getObjectById (eieio);
String name = emp.getName();
```

Persistent Object Model

The Java execution environment supports different kinds of classes that are of interest to the developer. Typically, application classes are highly interconnected, and the instances of those classes include the entire contents of the database.

Applications typically deal with a small number of persistent instances at a time. Transparent Persistence creates the appearance that the application can access the entire graph of connected instances, while in reality only a small subset of instances needs to be instantiated in the JVM.

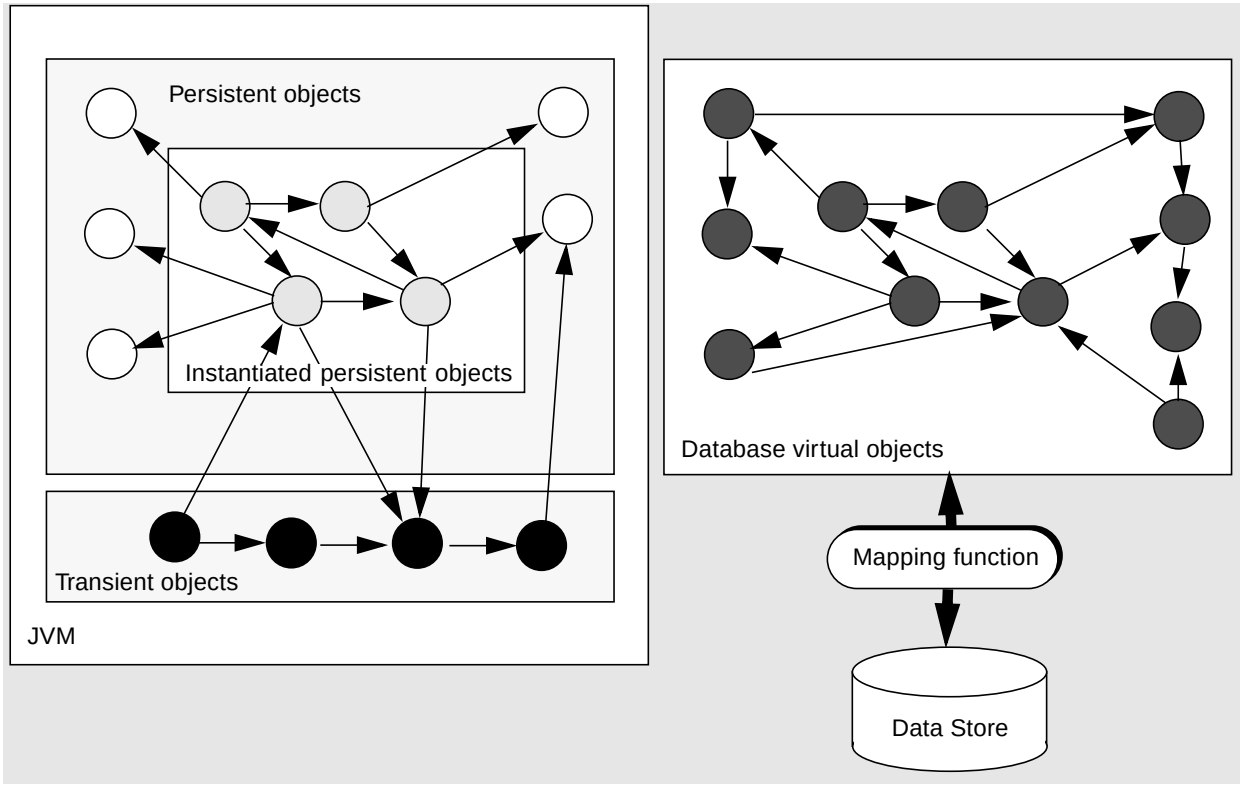

FIGURE 5-3 Instantiated Persistent Objects

Within a JVM, there can be multiple independent units of work that must be isolated from each other. Transparent Persistence permits the instantiation of the same database object into multiple Java instances. Whenever a reference is followed from one persistent instance to another, Transparent Persistence instantiates the required instance into the JVM.

The storage of objects in databases is different from the storage of objects in the JVM. Transparent Persistence creates a mapping between the Java instances and the objects in the database, using metadata that is available at runtime.

There is no restriction on types of non persistent fields of persistence-capable classes. These fields behave exactly as defined by the Java language. Persistent fields of persistence-capable classes have restrictions in Transparent Persistence, based on the characteristics of the types of the fields in the class definition.

Architecture

In Java, variables (including fields of classes) have types. Types are either primitive types or reference types. Reference types are either classes or interfaces. Arrays are treated as classes.

Instances are of a specific class, determined when the instance is constructed. Instances may be assigned to variables if they are assignment-compatible with the variable type.

The Transparent Persistence object model distinguishes between two kinds of classes: those that are persistence-capable and those that are not. User-defined classes are persistence-capable unless their state depends on the state of inaccessible or remote objects (for example, if they extend `java.net.SocketImpl` or implement their behavior by using native calls).

System-defined classes (those defined in `java.lang`, `java.io`, `java.net`, and so on) are not persistence-capable, nor are they allowed to be any of the following persistent field types:

- All primitive types (boolean, byte, short, int, long, char, float and double)
- All immutable object class types (Boolean, Character, Integer, Long, Float, Double and String as Second Class Objects)
- Mutable object class types from the `java.util` package (Date, ArrayList, and Vector) and mutable object class types from the `java.sql` package as Mutable Second Class Objects (Date, Time, Timestamp)

Persistent and Transient Objects

Classes associated with a database are designated as persistence-capable classes. Objects representing these classes can be either persistent objects or transient objects. Persistent objects are stored in a database. Transient objects exist only for the duration of the program that instantiates them.

All classes whose instances can be stored in a database must implement the `PersistenceCapable` interface. Transparent Persistence automatically adds the implementation of this interface when it enhances Java classes.

Field Types of Persistent-Capable Classes

In persistence-capable classes, fields can be persistent, transactional non persistent, or nontransactional non persistent.

Persistent Fields

TABLE 5-10 describes the persistent field types.

TABLE 5-10 Persistent Field Types

Field Type	Description
Primitive	Transparent Persistence supports fields of any of the primitive types `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`. Primitive values are stored in the database associated with their owning First Class Object. They have no Transparent Persistence Identity.
Immutable Object Class	Transparent Persistence supports fields of immutable object classes and can choose to support them as Second Class Objects or First Class Objects. `package java.lang:` `Boolean`, `Character`, `Integer`, `Long`, `Float`, `Double`, and `String` Transparent Persistence applications should not depend on whether these fields are treated as Second Class Objects or First Class Objects.
Mutable Object Class	Transparent Persistence supports fields of mutable object classes and may choose to support them as Second Class Objects or First Class Objects. `package java.util:` `Date` and `HashSet` `package java.sql:` `Date`, `Time`, and `Timestamp`. Because the treatment of these fields might be as Second Class Objects, the behavior of these mutable object classes when used in a persistent instance is not identical to their behavior in a transient instance.
`PersistenceCapable` Class	Transparent Persistence supports fields of `PersistenceCapable` class types as First Class Objects.
Collection Interface	Transparent Persistence supports fields of interface types. `package java.util:` `Collection` and `Set`

Persistent and Non-Persistent Fields

A persistence-capable class can have both persistent fields and non-persistent fields.

- Persistent fields are used to represent persistent data, and the Transparent Persistence runtime environment manages them for users of the class. This means that the Transparent Persistence runtime environment will automatically synchronize a persistent field's value with the database, flush object values to the database, and so on, in accordance with current transaction status, and concurrency management strategy.
- Non-persistent fields are managed by application logic; they do not participate in the Transparent Persistence mechanism. The application can use them for values that are derived from persistent values, values used in a transaction that do not need to be saved to the database, and so on.

JDO Interfaces

The JDO interfaces, found in a package named `com.sun.forte4j.persistence`, are:

- `PersistenceManagerFactory`–Allows users of Transparent Persistence classes (application developers) to create Persistence Managers.

  Developers cannot use Persistence Manager constructors, but use a Persistence Manager Factory to create a Persistence Manager. The Transparent Persistence API includes a class that implements this interface. The application instantiates the Persistence Manager Factory, configures its properties, and then creates a Persistence Manager. Any Persistence Manager Factory property settings become default settings for Persistence Managers created by the factory. If you want to use connection pooling, the Persistence Manager Factory can be used to set these properties as well.

- `PersistenceManager`–Manages and manipulates persistence-capable classes (which results in database selects, insert, updates, deletes) in transactional mode.

  The Persistence Manager normally manages all interactions with the database, including refreshing cached copies of persistent data. The application needs only to identify transaction boundaries.

  Each Persistence Manager manages a set of persistence-capable class instances created by the application, or that the Persistence Manager fetches in response to a query constructed by the application. Each Persistence Manager is capable of one transaction. In other words, a Persistence Manager generally manages a set of persistent instances created or fetched by a single client session, and each client session generally requires its own Persistence Manager. A Persistence Manager can connect to only one database (it can use multiple tables from that database), so some client sessions will need to obtain more than one Persistence Manager from more than one Persistence Manager Factory.

- `Transaction`–Allows users of persistence-capable classes to start and commit or roll back transactions.

  Developers obtain an object that implements this interface from the Persistence Manager. Transaction boundaries apply to persistent instances that are managed by that Persistence Manager. If the application is performing multiple database transactions, they must use multiple Persistence Managers.

- `Query`–Allows users of persistence-capable classes to construct queries.

  Developers obtain an object that implements this interface from the Persistence Manager, then use Query methods to construct a query in JDO query syntax. Completed queries can be executed by calling their `execute()` methods. Results are returned to the application as a collection of instances of a Transparent Persistence class.

- JDO exceptions–The JDO specification defines `JDOException` and a number of other exceptions derived from it. These are unchecked runtime exceptions. Application developers should code to catch those JDO exceptions their application might throw.

Transparent Persistence includes a `.jar` file that contains the implementations of these interfaces. The Persistence Manager Factory is implemented as a class that developers can instantiate directly; the other objects will be obtained by calling the appropriate factory methods.

By definition, a persistence-capable class is one that implements the `PersistenceCapable` interface. This interface provides a set of methods that allow users of Transparent Persistence classes (application developers) to check the status of Transparent Persistence instances.

Transparent Persistence classes must implement this interface, but the class developer does not write the implementation code. Instead, it is generated by Transparent Persistence during enhancement. After a class has been enhanced, it is able to interact with the Transparent Persistence runtime environment. Neither the developer of persistence-capable classes nor the application developer who uses them needs to be aware of what is in the generated code that implements the `PersistenceCapable` interface.

Transparent Persistence classes can be portable, which means that they can be moved from one JDO environment to another, be enhanced again in the new environment, and operate properly.

JDO Exceptions

TABLE 5-11 summarizes the exceptions associated with the rule violations.

TABLE 5-11 JDO User Exceptions

Exception	Explanation
`JDOException` (“Object is not `PersistenceCapable`”)	You cannot make an object persistent from a class that does not implement `PersistenceCapable`.
`JDOUserException` (“An instance with the same primary key already exists in this PM cache”)	You cannot use makePersistent on a different Java object with the same database identity.
`JDOFatalUserException` (“PM is closed”)	You cannot access a closed Persistence Manager.
`JDOFatalInternalException`	There has been an unexpected error at mapping or runtime.
`JDOUnsupportedOptionException`	You cannot use an unsupported option (for example, `setIgnoreCache(false)`).
`JDODataStoreException`	There is a conflict in the database or an integrity constraint violation.
`JDOQueryException` (“Missing candidate class specification.”)	The candidate class not specified. See the Query method `setClass`.
`JDOQueryException` (“Missing candidate collection specification.”)	The candidate collection not specified. See the Query method `setCandidates`.
`JDOQueryException` (“Candidate collection does not match candidate class <class>.”)	The candidate collection is not the extent collection from the candidate class.
`JDOQueryException` (“Wrong number of arguments.”)	There are more actual parameters passed to executes than are defined in `declareParameters`.
`JDOQueryException` (“Unbound query parameter ’param’.”)	The Query method `execute` does not get a value for the Query parameter ’param’.
`JDOQueryException` (“Incompatible type of actual parameter. Cannot convert ’java.lang.String’ to ’long’.”)	The type of the actual parameter is not compatible with the type in the parameter declaration.

TABLE 5-11 JDO User Exceptions *(Continued)*

Exception	Explanation
`JDOQueryException` (“<method> column(<nr>): <problem description>.”)	This form indicates a problem with the Query definition. <method> is one of the Query methods (`setFilter`, `declareParameters`, `setOrdering`, and so on). <nr> is the column number of the error. <problem description> is a description of the error, such as Syntax error or Invalid arguments(s) for ’<’. For example, the filter expression "`this.michael == 0`" would result in a `JDOQueryException("setFilter column(6): Field ’michael’ not defined for class ’com.xyz.hr.Employee’.")`, if the class `Employee` does not define a field `michael`.

Debugging Persistence-Aware Applications

The Persistence Debugger lets you debug persistence-aware applications without the need to package the persistence-capable classes as a JAR file. Like the Persistence Executor, the Persistence Debugger uses a special classloader to apply the enhancement of the classfiles for Transparent Persistence when they are loaded.

▼ To Debug an Application

1. **Make sure the JDBC driver is mounted or listed in your CLASSPATH.**
2. **Open the application within the IDE debugging environment.**
3. **Select Project > Settings**
4. **Choose Debugger Types, then choose Persistence Debugger.**
5. **Use the Persistence debugger as you would any other Forte for Java debugger.**

   For more information on using the Forte for Java debugging environment, see “Debugging a Program” in the Core IDE online help.

CHAPTER 6

Using Transparent Persistence With Enterprise Java Beans

This chapter describes how you can use Transparent Persistence with Enterprise Java Bean components, and includes sample code for using persistence-capable classes with J2EE™ Reference Implementation (J2EE RI) and iPlanet™ Application Server (iAS) applications.

Note – Forte for Java does not provide for deployment of Enterprise JavaBeans that use Transparent Persistence that were developed outside the Forte for Java IDE.

How Transparent Persistence Works in Enterprise Beans

Transparent Persistence provides you with an object view of persistent data stored in relational databases. The persistent instances can be used in the Enterprise Beans environment as helper objects with session beans or entity beans. You can use Transparent Persistence to improve performance in an enterprise bean so that it does not need to access the database as frequently. Instead of coding separate `get` and `set` methods, you can use serialized persistence-capable classes as value objects that display and update multiple fields.

To use Transparent Persistence in an Enterprise Beans environment, first develop persistence-capable classes as you would for any persistent application. Once these classes have been developed, they can be used with Enterprise Beans.

When you use persistence-capable classes with Enterprise Beans, the environment is slightly different compared to use in a two-tier application. These differences have to do with how the PersistenceManager is obtained and how transactions are managed:

- While the bean instance is activated, you make a JNDI lookup call to find the Persistence Manager Factory.
- The EJB container and Transparent Persistence coordinate transaction management in a persistence-aware enterprise bean that doesn't manage its own transactions.

An enterprise bean is somewhat different, too, when it uses Transparent Persistence. The main differences are how business methods are implemented, how synchronization is handled, and how transactions can be managed.

- Your bean's business methods are implemented by using a reference to an instance of a persistence-capable class that accesses and modifies the bean's state as required.
- Transaction synchronization is handled by the Persistence Manager, when the container makes transaction-completion call-backs at appropriate points in the bean's life cycle.
- Each business method must acquire its own `PersistenceManager` instance from the `PersistenceManagerFactory`. At the end of the business method, the `PersistenceManager` instance must be closed. This allows transaction synchronization between Transparent Persistence runtime and the container.
- A bean can use container-managed transactions for transaction completion. In that case, there is no extra code to be added. Or, if the bean manages its own transactions, it can use either a user transaction (that is, an instance of the `javax.transaction.UserTransaction` interface) or a Transparent Persistence transaction that it acquires from the Persistence Manager.

Session and Entity beans acquire a PersistenceManager, and with it can perform CRUD (create, read, update, and delete) operations on persistent instances using the interfaces defined in PersistenceManager, exactly as if the application were running in a two-tier environment.

To locate persistent instances to use in business methods, call the `getObjectByID(Object oid)` method of the `com.sun.forte4j.persistence.PersistenceManager` interface, or execute a query using the `com.sun.forte4j.persistence.Query` interface. This is no different from two-tier applications.

A typical sequence for using Transparent Persistence with Enterprise Java Beans is:

- Develop or map persistence-capable classes within the Transparent Persistence environment.
- Use the IDE's EJB Builder wizard to generate an enterprise bean, with code that does the following:
  - Generates a dynamic JNDI lookup for `PersistenceManagerFactory`
  - Acquires `PersistenceManager` using a `getPersistenceManager()` call to the `PersistenceManagerFactory`
  - Performs desired operations with persistent data
  - Closes `PersistenceManager`

If you want use connection pooling, you must configure the datasource properties outside of the Persistence Manager Factory.

Enterprise Java Beans, J2EE RI, and iAS are described in detail in the documentation included with their respective modules. You can also find White papers with more examples of how Transparent Persistence is used with Enterprise JavaBeans at the Forte for Java Portal as they become available.

Providing for Serialization

If you intend to pass a persistence-capable class as a parameter or return type of an EJB method, you must make the class serializable. When you generate persistence-capable classes, you have the option of defining them to be serializable (that is, implementing the `java.io.Serializable` interface).

When you pass an object outside the virtual machine that hosts its Persistence Manager, the Persistence Manager can no longer track the object's state. Therefore, if you want the enterprise bean's client to be able to update an object that it has received from the bean's remote interface, you must provide a bean method that accepts the modified object and applies the changes to the persistent instance. Or, you can have the client decide on the changes to be made and use another bean method to update the information.

Here's another case. Within a business method, a persistence-capable instance might refer to another persistence-capable instance that is part of the transaction but has no associated enterprise-bean component. If such a reference must be returned to the client, be sure that the instance's class is serializable.

For example, your entity bean `OrderBean` uses the persistence-capable class `Order` as a helper instance, and `Order` uses the persistence-capable class `LineItem`.To return an array of persistence-capable `LineItem` instances, you make the `LineItem` class serializable, and you write a remote method on `OrderBean` with the following signature:

```
public Collection getLineItems()
```

To create a serialized copy of a persistence instance, use the JDOHelper method `createSerializedCopy` and call it before the call to close `PersistenceManager`. This is illustrated in the following example.

```
persistenceManager = persistenceManagerFactory.getPersistenceManager();
//perform the query or navigation
//Collection items = ...
Collection result = (Collection)JDOHelper.createSerializedCopy(items);
persistenceManager.close();
return result;
```

Transactions With Enterprise Beans

Transaction management is the process of telling the container when to begin a transaction, and when to end it, as well as whether the transaction is to be committed or rolled back. With enterprise beans, transaction management is handled in a standard way that varies based on the kind of bean.

When programming Entity Beans and Session Beans with Container Managed Transaction completion, application components never complete transactions. When programming Session Beans with Bean Managed Transaction completion, the bean is responsible for completing transactions.

Regardless of which type of bean you are using, Transparent Persistence will coordinate with the transaction completion semantics of the container.

The PersistenceManager is a transactional object. That is, it contains information specific to a particular transaction. The PersistenceManagerFactory manages a pool of PersistenceManagers, each of which might be associated with a different transaction. It is important for the bean to get the appropriate PersistenceManager for the transaction, by getting the PersistenceManager when the thread of execution is associated with the transaction.

Each business method should get the `PersistenceManager` from the `PersistenceManagerFactory`, and close it at the end of the business method.

For Stateful Session Beans with Bean Managed Transactions, it is a bean decision when to get the `PersistenceManager`, because the `PersistenceManager` might be managed as a conversational state.

Creating an Enterprise Bean That Uses Transparent Persistence

The following sections take you through the general process for creating an enterprise bean that uses persistent-capable classes. The sections assume you have already created your persistent-capable classes.

Setting the JNDI Lookup

With Enterprise Java Beans (Enterprise JavaBeans), every component that uses resources needs to identify those resources in the deployment descriptor, and dynamically obtain them by lookup in JNDI at runtime. JDBC Connections are an example of resources that are managed by the container and looked up by the bean components. In Transparent Persistence, the Persistence Manager Factory is a resource that needs to be configured as the deployment descriptor, and looked up at runtime.

The recommended approach is to declare the `PersistenceManagerFactory` reference in `java:comp/env/jdo/`*`persistencemanagerfactoryname`*.

When the corresponding name is given to the `InitialContext` at runtime, the container finds the appropriate Persistence Manager Factory and returns it to the bean.

The Persistence Manager Factory is the resource that is shared among many beans, and is associated with a JDBC DataSource. With enterprise beans, all the beans that use the same DataSource should share the same PersistenceManagerFactory. This allows different beans in the same transaction to find the same Persistence Manager.

During bean development, you identify the Persistence Manager Factory to be used by name. During deployment, the name is associated with a specific `PersistenceManagerFactory`. At runtime, the named Persistence Manager Factory is found by looking up the name with JNDI.

▼ To Perform a JNDI Lookup

In the bean, put the following variables:

```
String persistenceManagerFactoryResourceName = "java:comp/env/jdo/pmfname";
PersistenceManagerFactory persistenceManagerFactory;
```

In the `setSessionContext` method, include the following code:

```
InitialContext initialContext = new InitialContext();
persistenceManagerFactory= (PersistenceManagerFactory)
initialContext.lookup(persistenceManagerFactoryResourceName);
```

You can use the Session Bean template to create a new session bean with the JNDI lookup code already added. Then you need only to replace a place holder for the JNDI name with the actual name and add required references to the bean's property sheet.

▼ To Create a Transparent Persistence-Aware Session Bean Using the IDE

1. **Choose File > New > Session Bean.**

   The EJB Builder wizard appears.

2. **Choose the type of bean you want: container-managed transactions (CMT bean) or bean-managed transactions (BMT bean).**

3. **Check the Use Transparent Persistence check box, and click Next.**

Note – If you are creating a stateful CMT bean, you will have the option to implement the `SessionSynchronization` interface. Do not check this option.

   The EJB Components pane appears.

4. **Continue with the template until you create your bean.**

   The Bean appears in the explorer window, as shown in FIGURE 6-1.

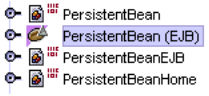

FIGURE 6-1 Persistent Enterprise Bean

5. **Right-click on the bean in the explorer window and choose Properties.**

6. **Select the J2EE RI tab.**

7. **Enter a value for the JNDI Name property.**

8. **Replace the generated JNDI name for the Persistence Manager Factory with the actual name.**

Setting Resource References

When setting up an Enterprise JavaBean with J2EE RI or iAS, you need to identify the Persistence Manager Factory as a Resource Factory Reference. You can do this through the Enterprise JavaBean property sheet.

▼ To Set the Persistence Manager Factory as a Resource Reference

1. **Right-click the Enterprise JavaBean node in the Explorer window.**
2. **Click on the value for Resource Factory References, then click on the ellipsis (...) button. A property editor opens.**
3. **Click on the Add button. The Add Resource Reference window opens.**
4. **Add the name of the Persistence Manager Factory.**
5. **Select com.sun.forte4j.persistence.PersistenceManagerFactory from the Type drop-down menu.**
6. **If you plan to deploy the application into the J2EE RI server, select the J2EE RI tab and enter the JNDI name exactly as in Step 4.**
7. **Click OK to finish.**

Note – If you are creating an Enterprise JavaBean with iAS, you will also need to add the reference to the Data Source in the Property sheet, using the same procedure as listed above.

Using Bean-Managed Transactions

You must decide whether to complete transactions by using the `javax.transaction.UserTransaction` supplied by the container, or the `com.sun.forte4j.persistence.Transaction` supplied by the `PersistenceManager`.

If you want to use the same PersistenceManager for multiple transactions, then you must complete transactions using `com.sun.forte4j.persistence.Transaction`. If you get a Persistence Manager for each transaction, it is your choice which technique to use.

To use `com.sun.forte4j.persistence.Transaction` for transaction completion, use the following code as an example:

```
// business method with multiple transactions with the same PersistenceManager
persistenceManager = persistenceManagerFactory.getPersistenceManager();
persistenceManager.currentTransaction().begin();
// perform persistent operations in the first transaction
persistenceManager.currentTransaction().commit();
PersistenceManager.currentTransaction().begin();
// perform persistent operations in the second transaction
persistenceManager.currentTransaction().commit();
persistenceManager.close();
```

If you use `javax.transaction.UserTransaction` for transaction completion, then you must begin the transaction before getting the PersistenceManager from the PersistenceManagerFactory, and close the PersistenceManager before you commit the transaction.

To use `javax.transaction.UserTransaction` for transaction completion, use the following code as an example:

```
sessionContext.getUserTransaction().begin();
persistenceManager = persistenceManagerFactory.getPersistenceManager();
// perform persistent operations in the first transaction
persistenceManager.close();
sessionContext.getUserTransaction().commit();
```

Using Container-Managed Transactions

When programming Entity Beans and Session Beans with Container Managed Transaction completion, application components never complete transactions. Transparent Persistence will coordinate with the transaction completion semantics of the container.

▼ To Use a Container-Managed Transaction

1. **In the bean, put the following variable:**

```
PersistenceManager persistenceManager;
```

2. **In each business method, wrap the following code around operations on persistent instances:**

```
persistenceManager = persistenceManagerFactory.getPersistenceManager();
// perform persistent operations
persistenceManager.close();
```

The following is an example of a container-managed transaction:

```
public java.lang.String addEmployee(long empid,
       java.lang.String lastName,
       java.lang.String firstName,
       double salary) {
    Employee emp = new Employee();
    emp.setEmpid(empid);
    emp.setLastname(lastName);
    emp.setFirstname(firstName);
    emp.setSalary(salary);

    try {
        persistenceManager =
persistenceManagerFactory.getPersistenceManager(dbuser, dbpasswd);
    persistenceManager.makePersistent(emp);
    return "Created Employee: " + emp.getEmpid();
    } catch (Exception e) {
        e.printStackTrace();
        return "Failed Create Employee: " + empid + ": " + e.toString();
    } finally {
        persistenceManager.close();
    }
}
```

Integrating Transparent Persistence Into the J2EE Reference Implementation

You can use Transparent Persistence with the J2EE RI (version 1.2.2 only) as described previously, with the following added procedures.

1. **Copy the JDBC driver and the following three files from the module installation directory to** *J2EE_HOME*`/lib/system`**:**
   - *IDE_Install*`/modules/dbschema.jar`
   - *IDE_Install*`/modules/ext/persistence-rt.jar`
   - *IDE_Install*`/lib/ext/xerces.jar`

   Note – *IDE_Install* will either be the IDE installation directory, or the Forte for Java (FFJ) user directory if you downloaded the FFJ module from the Update Center.

2. **Edit** *J2EE_HOME*`/bin/userconfig.sh` **script to add the JDBC driver and above three jars to the J2EE_CLASSPATH.**

3. **Set the JNDI lookup for the Persistence Manager Factory and your datasource.**

   a. **Edit** `J2EE_HOME/config/default.properties`**.**

      - Add `com.sun.forte4j.persistence.internal.EJB.j2sdkee121Helper` to the list of drivers to be loaded at the server startup time to enable the integration:

```
jdbc.drivers=...:com.sun.forte4j.persistence.internal.ejb.j2sdkee121Helper
```

      - Register `PersistenceManagerFactory` as a data source, replacing `jdo/`*empPMF* and `jdbc/`*datasource* with your own settings of JNDI names for the Persistence Manager Factory and DataSource:

```
jdbc20.datasources=jdo/empPMF|
xadatasource.0.jndiname=jdo/empPMF
xadatasource.0.classname =
  com.sun.forte4j.persistence.PersistenceManagerFactoryImpl
xadatasource.0.prop.ConnectionFactoryName=jdbc/datasource
xadatasource.0.prop.Optimistic=false
```

Note – Make sure there are no trailing spaces or non-displayed characters in the above lines, because the J2EE server will not recognize them. Also, verify that the value for transaction.timeout is set to "0".

b. Edit the J2EE RI startup script to add a system property.

By default, Transparent Persistence requires that each persistence-capable class be loaded by only one class loader. The effect of this standard behavior with J2EE RI is that persistence-capable classes can only be used with one J2EE application, and redeployment of the application is not possible. Adding the following system property to the J2EE RI startup script changes this default behavior.

```
-Dcom.sun.forte4j.persistence.model.multipleClassLoaders
```

c. Give the system property one of the following values:

- `ignore`: Use only one persistence-capable class definition. This setting is suitable where the same persistence-capable class is used in multiple applications and the class definition is identical in each one, or where you are using the persistence-capable class in only one application, and are modifying it during the develop/deploy/test cycle.
- `reload`: Replace the existing persistence-capable class definition. This setting is suitable where you are using the persistence-capable class in only one application, and are modifying it during a develop/deploy/test cycle.

In a Solaris environment, set the PROPS variable in `$J2EE_HOME/bin/j2ee` to:

```
PROPS="-Dcom.sun.enterprise.home=$J2EE_HOME -
Djava.security.policy==$J2EE_HOME/lib/security/server.policy
-Dcom.sun.forte4j.persistence.model.multipleClassLoaders=reload"
```

In a Windows environment, directly change the `%JAVACMD%` command in `j2ee.bat`:

```
%JAVACMD% -Djava.security.policy==%J2EE_HOME%\lib\security\
server.policy -Dcom.sun.enterprise.home=%J2EE_HOME% -
Dcom.sun.forte4j.persistence.model.multipleClassLoaders=reload -
classpath "%CPATH%" com.sun.enterprise.server.J2EEServer %1 %2
```

4. Start the J2EE RI server, and create your Enterprise JavaBean as described in the Enterprise JavaBean and J2EE RI documentation.

You will also need to set the PersistenceManagerFactory as a Resource reference, as described in “Setting Resource References” on page 147.

Note – If you need to do a rollback in J2EE RI business methods that use container-managed transactions for transaction demarcation, you must prepare a serialized copy of a persistence instance before calling `ctx.setRollbackOnly()`. See "Providing for Serialization" on page 143.

Integrating Transparent Persistence With the iPlanet Application Server

The iPlanet Application Server (iAS) 6.0 SP3 plug-in module provides an application program interface (API) for the iPlanet application and web server plugin modules. You can use Transparent Persistence with the iAS as described in Using Transparent Persistence with Enterprise JavaBeans with the following added procedures:

1. **Change registry parameters to be able to register and perform a JNDI lookup of the `persistenceManagerFactory` resource reference.**

   a. **Run `kregedit` (located at `IAS_Install_dir/ias/bin/kregedit` on Solaris or `kregedit.bat` on Windows.)**

   b. **Click on SOFTWARE\iPlanet > Application Server 6.0 > jndiConfig.**

   c. **Select Edit >Add Key, then type `jdo`.**

   d. **Right-click on `jdo`.**

   e. **Select Edit >Add Value. Add the following:**
      - `contextClassName`
      - `com.netscape.server.jdo.PMFContext`

   f. **Select Edit > Add Value again. Add the following:**
      - `factoryClassName`
      - `com.netscape.server.jdo.PMFContextFactory`

2. **Add the necessary JAR files to the CLASSPATH.**

   a. **On Solaris:**

- Insert the following code before the `THIRD_PARTY_JDBC_CLASSPATH` line in `IAS_Install_dir/ias/env/iasenv.ksh`:

```
FFJ_IDE=IDE_Install TP_PATH=$FFJ_IDE/modules/dbschema.jar:
$FFJ_IDE/lib/ext/xerces.jar:$FFJ_IDE/modules/ext/persistence-
rt.jar:$FFJ_IDE/iPlanet/jdoias/iaspmf.jar:$FFJ_IDE/iPlanet/jdoias
```

- Add the Transparent Persistence path `$TP_PATH` in front of the CLASSPATH: `CLASSPATH=$TP_PATH:`*existing code*

b. In a Windows environment, you'll need to edit the Java CLASSPATH:

- Select SOFTWARE\iPlanet Application Server 6.0 Java CLASSPATH registry and add the following in front of the path:

```
IDE_Install/modules/dbschema.jar:IDE_Install/lib/ext/xerces.jar:IDE_Install/mo
dules/ext/persistence-rt.jar:IDE_Install/iPlanet/jdoias/iaspmf.jar:
IDE_Install/iPlanet/jdoias
```

3. Restart the iPlanet Application Server.

Follow the steps in the documentation to enable the iPlanet plugin.

4. Add and register PersistenceManagerFactory.

a. Click on JDO(TP) Persistence Manager Factories, choose Add a Persistence Manager Factory, and fill in values for the properties:

```
Connection Factory = jdbc/PointBase
//("jdbc/" + DataSource name)
Persistence Manager Factory Name = empPMF
//(other boolean settings are optional)
```

b. Right-click on the created Persistence Manager Factory (`empPMF`) and choose Register. Choose your server in the Select Server to Register window. Press the Register button.

c. Set the PersistenceMangerFactory as a Resource Factory using the Enterprise JavaBean Property sheet.

This is described in "Setting Resource References" on page 147. You also need to set DataSource reference for all Enterprise JavaBeans that use `PersistenceManagerFactory`.

5. Follow steps in the plug-in documentation to fix resource references for iAS deployment of your beans.

APPENDIX **A**

System Requirements

Transparent Persistence supports development and use of persistence-capable classes with the DB2 Universal Database, Oracle 8*i*, PointBase, and Microsoft SQL Server.

In addition to the Transparent Persistence module, running in Forte for Java, you need one of the following supported JDBC drivers installed in the `lib/ext` subdirectory of the Forte for Java installation directory:

- WebLogic for SQL Server 2000 driver
- PointBase Embedded 3.5 driver with PointBase bundled in IDE
- ORACLE 8*i* 8.1.6 Thin
- DB2 Universal Database, Version 7.1

Note – Transparent Persistence depends on ANTLR 2.7.0 in order to parse query statements. ANTLR 2.7.0 is included, and works automatically, but will conflict with other versions of ANTLR you may have in your runtime JVM. Be sure to disable any other versions of ANTLR before running Transparent Persistence.

Your `CLASSPATH` variable needs to include the following software:

- A supported JDBC driver
- Transparent Persistence runtime package, `persistence-rt.jar`
- `dbschema.jar` from the `modules` directory of the Forte for Java installation, from <*FFJ install root*>`/modules/dbschema.jar`
- An XML SAXParser from <*FFJ install root*>`/lib/ext/xerces.jar`

The location of the persistence-rt.jar and dbschema.jar files depend on how you install the Transparent Persistence and DBSchema modules:

- If you install the modules at the same time you install the IDE, the files will be located in <*install root*>`/modules/ext`.
- If you install the modules from the Update Center running in multi-user mode (the default), they will be in <*ffjuser*>`/modules/ext`.

- If you install the modules from the Update Center in single user mode, they will be in <*install root*>/modules/ext.

Note – If you are running Transparent Persistence when you modify your CLASSPATH variable, you will need to restart Forte for Java for the changes to take effect.

APPENDIX **B**

Transparent Persistence JSP Tags

Transparent Persistence supports the JSP tags `PersistenceManager` and `jdoQuery`. For general information on JSP tags, refer to *Building Web Components* in the Forte for Java Programming Series.

`PersistenceManager` Tag

The `PersistenceManager` tag creates a `PersistenceManager` that is used by the `jdoQuery` tag to retrieve objects through a `database.jdbc` connection. You can store the Persistence Manager in any of the four scopes: application, session, request or page. The default scope is application.

`PersistenceManager` attributes:

- `id` (required)

  The ID under which the `PersistenceManager` information is stored. The JDO Query tag uses id to retrieve the objects. The attribute can be set statically or using a JSP expression.

- `scope`

  The scope where the `PersistenceManager` is stored. The value needs to be application, session, request, or page. The attribute can be set statically or using a JSP expression.

- `connection` (required)

  The attribute specifies the connection ID, which is used to retrieve the connection information. The attribute can be set statically or using JSP expression.

- `connectionScope`

The scope where the connection ID is searched. The value needs to be application, session, request, or page. If the attribute is not specified, the system searches all the scopes in the following order: page, request, session, application. The attribute can be set statically or using JSP expression.

`PersistenceManager` Tag Example:

```
<%@taglib uri="/WEB-INF/lib/dbtags.jar" prefix="jdbc" %>
<%@taglib uri="/WEB-INF/lib/tptags.jar" prefix="jdo" %>
<jdbc:connection id="conn"
 driver="weblogic.jdbc.mssqlserver4.Driver"
 url="jdbc:weblogic:mssqlserver4:marina@bete:1433"
 user="mv" password="mv" />
<jdo:persistenceManager id="empPM" connection="conn" />
```

`jdoQuery` Tag

The `jdoQuery` tag is used to query the database and get the results. These results then can be passed to iterator tags in order to be displayed.

The `jdoQuery` tag supports the standard SQL statements Insert, Update, Delete and Select. Because the SQL statement is specified in the body instead of as an attribute, JSP scripting can be used to control how query is created.

`jdoQuery` attributes:

- ID (required)

  The ID under which the query instance is stored. If a `queryid` instance is present in the scope specified by queryscope, then the body of the query is not executed. Note that `queryid` is different from the `ResultsId`. `ResultsId` is the ID under which the results are stored.

- `className` (required)

  Fully qualified class name (`package.subpackage.ClassName`) of the Object that will be retrieved from the database.

- filter

  The filter (for example, `emp.salary < 10000`) used to construct query to retrieve the Objects from the database.

- imports

  The import string that will be used to resolve the class names and variables used in the constructed query.

- variables

  The variables that will be used in constructing the query for retrieving the objects from database.

- `persistenceManager` (required)

  The `PersistenceManager` id used to construct and execute the query.

- `persistenceManagerScope`

  The scope where the `PersistenceManager` ID is searched. The value needs to be one of the following: application, session, request, page. If the value is not set, the system searches all the scopes in the following order: page, request, session, application. The attribute can be set statically or using JSP expression.

- `resultsId` (required)

  The result data from the query is stored under the value specified by this attribute. The attribute can be set statically or using JSP expression.

- `resultsScope`

  The scope where the result data is stored. The value specified should be one of the following: application, session, request, page.

`jdoQuery` Tag Example:

```
<%@taglib uri="/WEB-INF/lib/dbtags.jar" prefix="jdbc" %>
<%@taglib uri="/WEB-INF/lib/tptags.jar" prefix="jdo" %>
<jdbc:connection id="conn"
 driver="weblogic.jdbc.mssqlserver4.Driver"
 url="jdbc:weblogic:mssqlserver4:marina@bete:1433"
 user="mv" password="mv" />
<jdo:persistenceManager id="empPM" connection="conn" />
<jdo:jdoQuery id="employeeQuery"
 persistenceManager="empPM"
 className="empdept.post.Employee"
 resultsid="employeeDS" resultsScope="session" />
<% printJDOQueryResults(pageContext,out,"employeeDS"); %>
<jdbc:cleanup scope="session" status="ok" />
```

APPENDIX C

Restrictions and Limitations

In this appendix, we discuss unsupported or restricted features, the ways database-specific behaviors and limitations can affect your use of Transparent Persistence and the results you might receive, and file migration information for developers who have created classes using previous versions of Transparent Persistence.

The issues covered in this section are:

- Unsupported features and restrictions
- Restrictions and limitations on the use of Transparent Persistence with the following:
  - PointBase 3.5 Network (Multi-User) Server Product, bundled with the IDE.
  - Oracle 8.1.6 Thin Driver
  - WebLogic JDBC driver 5.1.0 for Microsoft SQL Server 2000
  - DB2 Universal Database, Version 7.1
  - The Microsoft JDBC-ODBC bridge
- Migrating classes created by earlier versions of Transparent Persistence

Unsupported Features

Transparent Persistence does not currently support the following features:

- Tables without primary keys
- The ability to update primary key values
- Join tables with extra columns
- User-defined concurrency groups
- User-defined, large object, and national character set datatypes, such as Blobs, Clobs, text, nChar, nVarchar, and ntext

- Inheritance: A persistence-capable class cannot extend directly or indirectly from another class.
- Relationships between classes across multiple database schemas
- Inserting and deleting object graphs containing circular dependencies
- Views that do not include all the primary key columns of the table (simple and composite primary keys). Transparent Persistence does not support views if they do not contain all the primary key columns.
- The runtime behavior of classes mapped to views is subject to the limitations of the underlying database with regard to updating and deleting views. If the limitations are violated, then the database will throw an exception. Some of these limitations include:
  - Views having aggregate functions (for example, `SUM, AVG, max, min, count, and count(*)`) in their definitions
  - Views having user-defined functions
  - Views having `WITH CHECK OPTION` in their definition
  - Views having the `group by` clause in their definition
  - Views having the `order by` clause in their definition

Restrictions

The following features are supported, but restricted in some cases.

Application Class Loaders

Transparent Persistence assumes that two persistent-capable classes that have a relationship are loaded using the same class loader. Transparent Persistence does not support the scenario that two classes having the same class name are loaded by different class loaders. This will result in a `JDOFatalUserException`, "class *class.Name* loaded by multiple class loaders".

In an application server environment, this restriction can be resolved by using the `com.sun.forte4j.persistence.model.multipleClassLoaders option`, as described in Chapter 6.

Comparing Collection Relationships

You cannot compare a collection relationship with a non-null value. The query will result in a `JDOUnsupportedOptionException`.

User-Defined `clone()` Methods

Transparent Persistence requires that a newly created clone of a persistence instance of a persistence-capable class is a transient instance with respect to Transparent Persistence. For almost all cases, this can be ensured by the Transparent Persistence's enhancer, which either generates an appropriate `clone()` methods (if none has been defined by the user) or adds some code to the byte-code of a user-defined `clone()` method.

The created clone is marked to be transient right after a generated or user-defined clone method has returned from calling the clone method of the superclass (`super.clone()`). Therefore, no user-defined clone methods in all superclasses of a persistence-capable class can directly or indirectly invoke code that accesses any persistent fields of the newly created clone. Such an invocation would cause an interaction with the Transparent Persistence runtime before the clone has been marked as transient in the persistence-capable subclass.

User-Defined Constructors

The Transparent Persistence runtime creates instances of a persistence-capable class using a special constructor that is added by the enhancer. This constructor does not call any other, constructors of the persistence-capable class (for example, user-defined constructors), but instead invokes a no-argument (also called the “default”) constructor of the superclass of the persistence-capable class. This imposes the following restrictions upon persistence-capable classes:

- The superclass must provide a default constructor accessible to the persistence-capable subclass.
- For persistence-capable classes, no user-defined constructors or initializations of non-static instance fields will be executed on instances created by the Transparent Persistence runtime as result of a query or relationship navigation.

Database Limitations and Restrictions

The following limitations and restrictions apply only to specific databases, as detailed below.

PointBase 3.5 Network (Multi-User) Server

This section describes how the PointBase Network Server 3.5, bundled with the IDE, behaves in certain circumstances.

Error Message: "`java.net.SocketException: Socket closed`"

If `PersistenceManagerFactory` is configured without connection pooling and there are several instances of the `PersistenceManagerFactory` created that are not in use any more, the garbage collection process prints the following message to the `System.out` when the connection is closed:

```
java.net.SocketException: Socket closed
```

The exception is ignored internally, so there is no affect on runtime.

PointBase Database version 3.4

Transparent Persistence cannot support the PointBase Database version 3.4 because PointBase Database version 3.4 does not support regular identifiers within quote marks.

To run your application with this version, you need to override 3.5 settings by creating a file `.tpersistence.properties` with the following two lines:

```
database.pointbase.QUOTE_CHAR_END=
database.pointbase.QUOTE_CHAR_START=
```

Place this file in the root directory of the application that calls the database.

`isEmpty()` Method

Using the isEmpty() method in a filter will throw a JDBC SQLException. An example of such a query is:

```
query.setFilter("employees.isEmpty()");
```

Location of PointBase Network Server

The Database Schema wizard assumes that the PointBase Network Server is located in the directory from which you started the database. For example, if you start the database from the IDE, it will assume the database is located in:

Forte_Home`\pointbase\network\databases`

Multiple Relationship Fields in a Fetch Group

You cannot put multiple relationship fields in a fetch group if you are using the PointBase Network Server.

Workaround: For each related field, make sure the Fetch Group property is set to `none`.

Oracle 8.1.6 Thin Driver

This section describes how the Oracle 8.1.6 database behavior can affect Transaction Persistence under certain circumstances.

Concurrent Transactions

Data store transactions with isolation level SERIALIZABLE behave differently in Oracle than they do with other supported databases. For example, note the following two transactions:

Transaction 1: Fetch an object into cache, then modify the object fields in the cache.

Transaction 2: Fetch an object with the same primary key values into another cache, then modify the object fields in the cache.

Most databases would put a read lock on the row and prevent you from committing transaction 2 before you commit transaction 1. Oracle, however, only blocks a transaction if another uncommitted transaction modifies the same row. If you try to

commit Transaction 2 before transaction 1, Oracle will commit transaction 2, then throw an exception, with the message `cannot serialize access for this transaction`.

Concurrent Update Operations

If you attempt concurrent update operations in a multi-threaded environment with Oracle, the process might hang.

Acquiring a Connection

The Oracle Thin Driver requires that a user name and password be specified when acquiring a connection. It can be specified when initializing a Persistence Manager Factory in a non-managed environment, or in a managed environment either when configuring the properties of a data source, or by providing non-null arguments to the method `PersistenceManagerFactory.getPersistenceManager(`*user, password*`)`.

WebLogic JDBC Driver 5.1.0 for Microsoft SQL Server 2000

This section describes how the WebLogic JDBC driver 5.1.0 for Microsoft SQL Server 2000 behavior can affect Transaction Persistence under certain circumstances.

One-to-One Relationships

An exception is thrown when you try to delete an instance that participates in One-to-one relationships if one of the foreign key columns has a unique constraint on it. Workaround: Null out the relationships in one transaction and then delete the instance in a new transaction.

J2EE Reference Implementation Application Server

If you are using the J2EE RI Application Server with the WebLogic driver, and the driver file is separate from the license file, you must repackage the license file into the driver file to have the `java.security.AllPermission` for all components of this driver.

DB2 Universal Database, Version 7.1

This section describes how the DB2 Universal database behavior can affect Transaction Persistence under certain circumstances.

One-One Relationships

You can not remove one-one relationships, or set them to “null” because DB2 adds a unique constraint to the Foreign Key column.

Columns With UNIQUE Constraints

Columns with UNIQUE constraints cannot have multiple null values.

DT_VARCHAR2_2000 Data Types

Transparent Persistence supports the DB2 DT_VARCHAR2_2000 data type, with the following restrictions:

- Do not put DT_VARCHAR2_2000 fields in the default fetch group. The query will fail. To prevent this, explicitly exclude JDBC Type DT_VARCHAR2_2000 fields from the default fetch group while mapping the table.
- You can only use DT_VARCHAR2_2000 fields in queries where the value of the DT_VARCHAR2_2000 field is compared to null. For example, `DT_VARCHAR2_2000Field == null`, or `DT_VARCHAR2_2000Field != null`.
- You cannot use Update and Delete operations during the commit of optimistic transactions. Use the Datastore transaction instead.
- DT_VARCHAR2_2000 fields can only be updated with entries that are less than or equal to 4000 bytes in size.

Select Statements

DB2 does not support queries that result in SELECT statement with `? <op> ?`. This can happen if you compare literals or query parameters.

Also, if a query attempts to select a record locked for an update, the application will hang. This can happen in a data store transaction when updated instances are flushed to the database prior to query execution.

Microsoft JDBC-ODBC Bridge

This section describes behavior of the Microsoft JDBC-ODBC Bridge (SQLSRV32.DLL) version 2.0001 (03.70.0623) that may affect Transparent Persistence:

Concatenation

Queries that concatenate strings (for example, queries with the filters `startsWith`, `endsWith`, or uses + (such as `"Engi" + "neering"`)) return 0 rows, but will not throw any exception.

Dates

Dates 2079-06-07 00:00:00.0 and higher fail for updates with the SQLException: `Datetime field overflow`.

Migrating Files

The file format for persistent classes has changed in this version of the Transparent Persistence module. Old files can be viewed and will run in this version, and can be used in an application that uses both old and new files. However, files in the new format will not work with older versions of the Transparent Persistence Module.

If you open and modify a previously created persistent file, Transparent Persistence will ask you if it can migrate your class to the latest format.

You can choose to:

- Save Now, which commits the modification and migrates the file immediately.
- Save Later, which migrates the file the first time the file is saved.
- Cancel, which cancels any modifications to the file and the file remains in its original format.

Index

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X