Programming Persistence

Forte™ for Java™, Internet Edition, 2.0

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

Part No. 806-7517-10
December 2000, Revision A

Contents

5 Developing Persistence-Aware Applications

A Transparent Persistence JSP Tags

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 product.

Who should 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

Before you read this book: The *Sun BluePrints™ Design Guidelines for J2EE* (`www.java.sun.com/j2ee/blueprints`) can help you understand the concepts upon which this tutorial is based.

Organization of This Manual

The following table briefly describes the contents of each chapter:

Chapter	Description
Chapter 1, “Overview of Persistence Programming”	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 Forte for Java.
Chapter 2, “Using Java Data Base Connectivity”	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, “Transparent Persistence Overview”	Provides a brief overview to the Transparent Persistence programming model.
Chapter 4, “Developing Persistence-Capable Classes”	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, “Developing Persistence-Aware Applications”	Describes the Transparent Persistence runtime environment and illustrates how to use it to perform persistence operations. It also addresses various Transparent Persistence programming issues.
Appendix A, “Transparent Persistence JSP Tags”	Documents two JSP tags that perform Transparent Persistence functions.

Conventions

This table provides information about the conventions used in this document.

Format	Description
italics	Italicized text represents a variable. Substitute an appropriate clause or value where you see italic text. Italicized text is also used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.
`monospace bold`	Monospace bold text represents user input contrasted with computer output.
ALL CAPS	Text in all capitals represents file system types (GIF, TXT, HTML and so forth), environment variables, or acronyms (FFJ, JSP).
Key+Key	Simultaneous keystrokes are joined with a plus sign. For example, Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen. For example, Esc-S means press the Esc key, release it, then press the S key.

Forte for Java, Internet Edition Documentation Set

Forte for Java offers a set of books delivered in Acrobat Reader (PDF) format and online help. This section provides descriptions of these documents.

Documentation Set

You can download the following documents from the Forte for Java web site:

- The Forte for Java programming series:
  - *Introduction*

    Introduces the two books in the Forte for Java, Internet Edition programming series.
  - *Building Web Components*

    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*

    Describes support for different persistence programming models provided by Forte for Java: JDBC and Transparent Persistence.
- *Forte for Java, Internet Edition Tutorial*

  Provides step-by-step instructions for building a simple web application using tools introduced in Forte for Java, Internet Edition, which facilitate creating a web module, as described in the *Java 2 Platform Enterprise Edition Specification*.

Online Help

Online help is available inside the Forte for Java development environment. Access it by pressing the Help button or F1 key, which opens a context-sensitive help window for the feature you are using. You can also choose Help > Contents from the Help menu, which gives you access to a list of help topics and a search facility.

Javadoc

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

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.

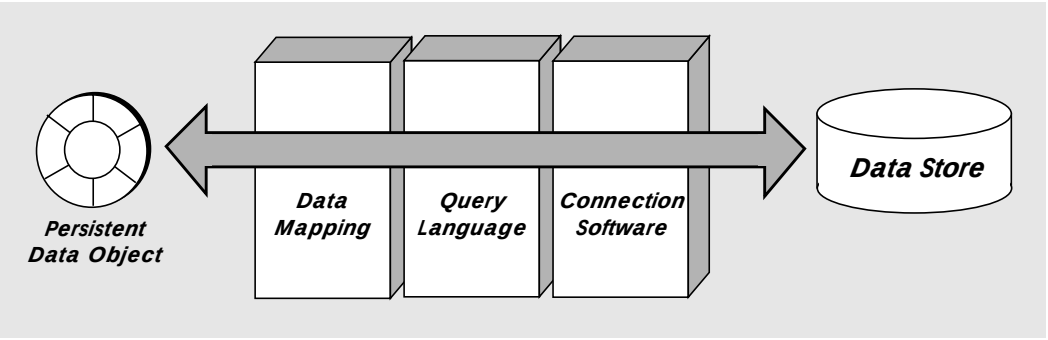


Figure 1 *Basic Persistence Scheme*

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) 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. JDBC 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 JDBC, described in Chapter 2.

JDBC Programming Model

The JDBC programming model follows closely the software layers identified in Figure 1 on page 8. 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 2 on page 11 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 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.

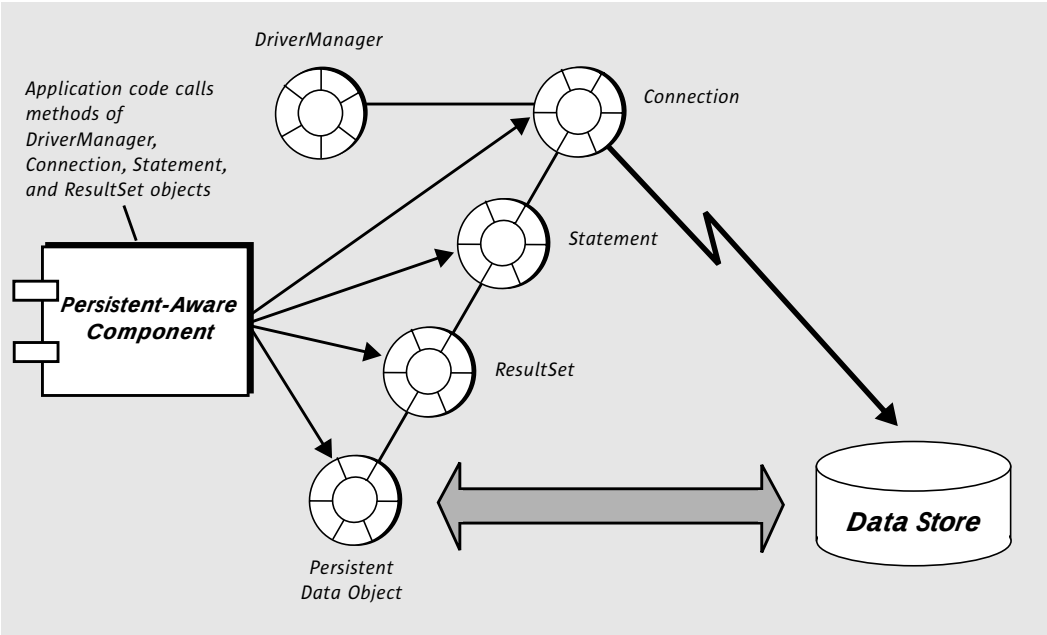


Figure 2 *JDBC Programming Model*

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 multithreaded; 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 multithreaded 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, and 5.

Transparent Persistence Programming Model

The Transparent Persistence programming model, unlike JDBC, automates most of the software layers identified in Figure 1 on page 8.

- 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, 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 3 on page 14 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 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.

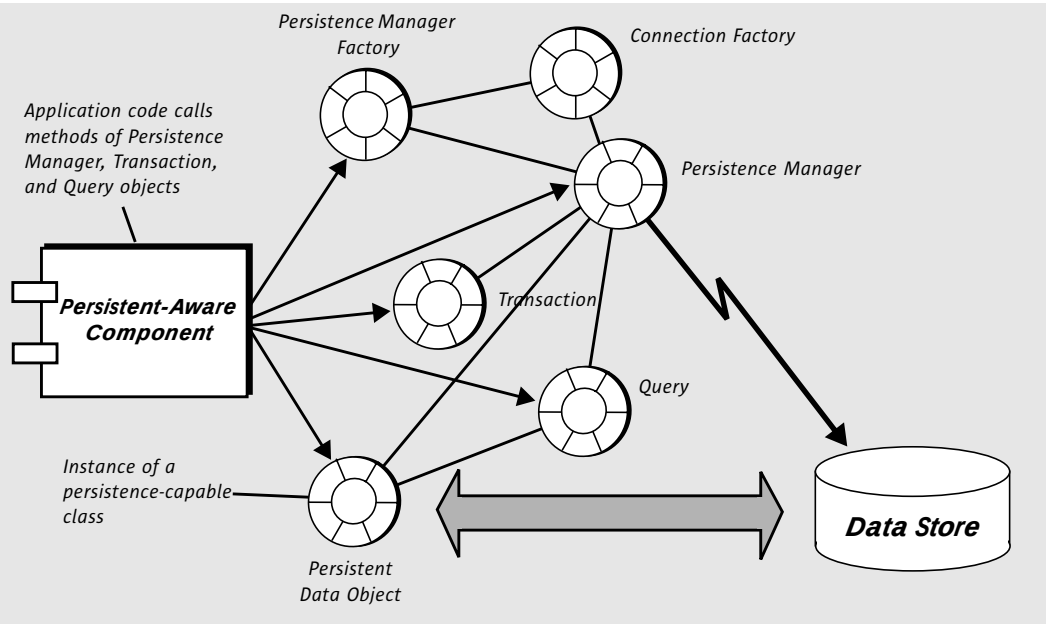


Figure 3 *Transparent Persistence Programming Model*

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.

Container-Managed Persistence

To support transaction management with standard components, including distributed transaction management, Enterprise Java Beans have a component model for managing transactional context on behalf of a user. Transparent Persistence was designed to integrate with Enterprise Java Beans in three areas:

- Use a Session Bean as the persistence-aware component, which obtains the Persistence Manager from the Persistence Manager Factory, and uses the Query and Transaction interfaces to obtain PC instances and manage transaction completion;
- Use a Bean-Managed Persistence Entity Bean to delegate to PC instances;
- Use a Container-Managed Persistence Entity Bean to manage persistent instances.

The integration with EJB is not a feature of FFJ 2.0, so it is not further described in this document. However, the persistence-capable components developed with FFJ 2.0 are intended to provide the persistence architecture for EJB integration.

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. Establishing 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 Using database tables.
   a. Create a table.
   b. Creating JDBC statements.
   c. Execute Statements to perform persistence operations.
      - Enter data into a table.
      - Obtain data from a table.
      - Create an updatable result set (RowSet).
      - 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/online Training/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 an reference when starting to program using JDBC:

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

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 lets 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.
- 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 `NBCachedRowSet`, which is a type of row group that contains information from the database—in the case of `NBCachedRowSet`, cached data. One type of non-visual component is a Data Navigator–a visual JDBC component that you add to a form to manipulate the display of data to the user.

To understand how to use JDBC JavaBeans 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`
- `NBCachedRowSet`
- `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 an NBCachedRowSet

An `NBCachedRowSet` component represents rows fetched from the database. Use this component 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, `RowSets` 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.

`RowSets` 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.

NBCached RowSet as a Type of RowSet

The JDBC tab provides for a `NBCached RowSet`, 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.

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

Table 1 *NBCached 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 NBCachedRowSet
Transaction isolation	Sets a transaction isolation level.

Expert and Event Tabs for an NBCachedRowSet

The Expert Tab for an NBCachedRowSet enables you to inspect and modify additional properties.

Table 2 *NBCachedRowSet Expert Tab Properties*

Property	Definition
Database URL	SQL query to populate this RowSet. The query can be any syntactically-correct SQL Select Query.
Default column values	Specifies whether default column values can be supplied for this NBCachedRowSet. You can supply default values through a Default Values Editor, a String Editor, a Resource Bundle, or a Form Connection. You can generate initialization code using the Advanced button. In the Default Values Editor, you can press Fetch columns to retrieve a list of columns in the NBCachedRowSet from which to obtain columns to select. As an alternative to obtaining metadata, you can enter the column name, or index in the rowset.
Execute on load	Specifies by a Boolean value whether the NBCachedRowSet 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.
Password	A password the user must supply to gain access to the table that contains this NBCachedRowSet.
Table Name	The name of a database table associated with this NBCachedRowSet. You can specify a table via a string editor, within a Resource Bundle, or from a Form Connection. You can also generate initialization code by using the Advanced button.
User Name	The name of a user associated with this table. You can specify a User Name via a string editor, within a Resource Bundle, or from a Form Connection. You can also generate initialization code by using the Advanced button.

The Event Tab for an NBCachedRowSet enables you to inspect and modify events associated with NBCachedRowSets.

Table 3 *NBCachedRowSet 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 NBCachedRowSet is changed.
rowInserted	Specifies event handlers for the rowInserted event.This method is called when a row in a NBCachedRowSet is inserted.
rowSetChanged	Specifies event handlers for the rowSetChanged event.This method is called when an NBCachedRowSet is changed.

Stored Procedure

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.

Table 4 *Stored Procedure Properties*

Property	Definition
Arguments	The arguments that are passed to the stored procedure.
Binded rowset for Stored Procedure	You can populate a RowSet with data after calling the stored procedure. Select the RowSet from a drop-down list on the Property Sheet.
Call format	The format in which the stored procedure is called. For example, the stored procedure might be called using Name and Arguments; these are also substitution codes for the properties with those names on this property sheet.
Connection provider	The configured connection source in whose context the stored procedure is called.
Name	The name of the called stored procedure.

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 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 automatically committed to the database.
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.

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.

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`), `NBCachedRowSet`, 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 `NBCachedRowSet` 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:

- `NBCachedRowSet`
- `Connection Source`
- `Pooled Connection Source`
- `Stored Procedure`

The `Data Navigator` component and other Swing components are showed 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 21.

The following sections illustrate the use of the JDBC Form Wizard.

Establishing a Connection

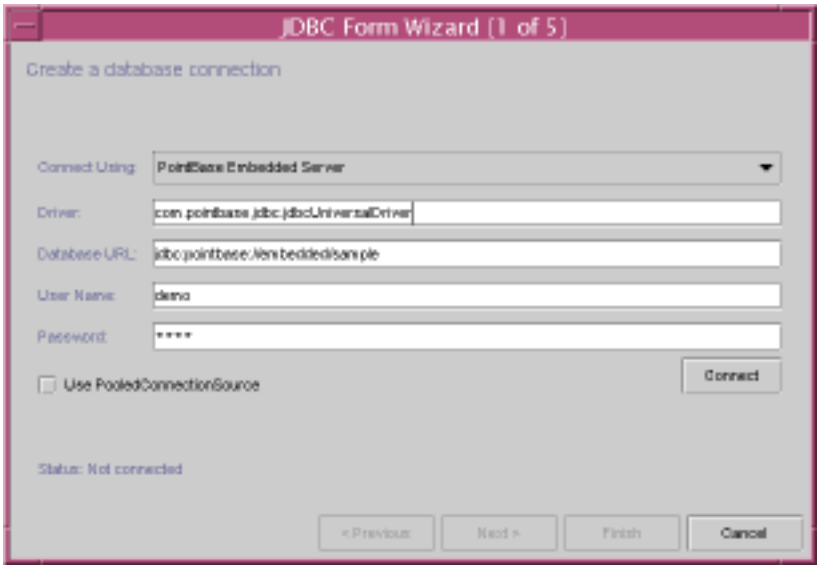
When you use the JDBC Form Wizard or when you use the JDBC tab to create a JDBC client application, one of the first tasks you must perform is to establish a connection with the database management system that you want to use.

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.

You invoke the JDBC Form Wizard from the Tools item on the Forte for Java Menu.

Once you invoked the JDBC Form Wizard, the first panel lets you establish a connection with a database. You can specify the use of a pooled connection for a Data Source.

Figure 4 *JDBC Form Wizard, Panel 1*



The `Active connections` combo box appears only if you have created database connections in the Database Explorer. It enables you to make a database connection by selecting a connection from this list. To make your choices easier, Forte for Java provides, through a template, a list of the most frequently used databases.

When you need a new connection, you must supply:

- Database URL
- Driver name
- User Name
- password

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

The `Status` field shows the status of the connection while you are using the JDBC Form Wizard.

When you click the `Connect` 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

The second panel of the JDBC Form Wizard lets you:

- Select a table 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 the section on "Transaction Isolation" in this document.
- 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 5, Forte for Java provides a default SQL command to use with the table you have selected.

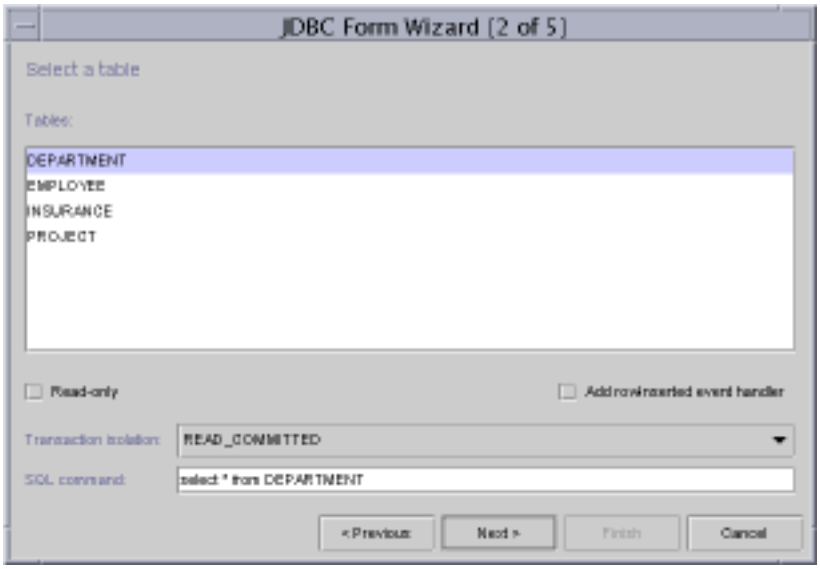


Figure 5 *JDBC Form Wizard, Panel 2*

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 6 *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.
REPEATABLE_READ	Prohibits a transaction from reading a row with uncommitted changes in. It also prohibits the situation where a transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, obtaining different values the second time.
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 third panel of the JDBC Form Wizard lets you select columns from the database tables to include in the form that is displayed. You can use this panel to manipulate the column display and to generate code that will be placed in your application.

This panel lets you select a view of the data to present to the user. In the example provided, `Jtable` (the most common Swing form) is used.

Other Swing component choices include:

- `Jlist`
- `JComboBox`
- `JTextField`

Note These components cannot display more than one column. When you choose one of these components to contain the data you want to display in your application, select a column in the `Displayed columns` box by clicking a value in the Name column and selecting a column name from the built-in combo box.

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

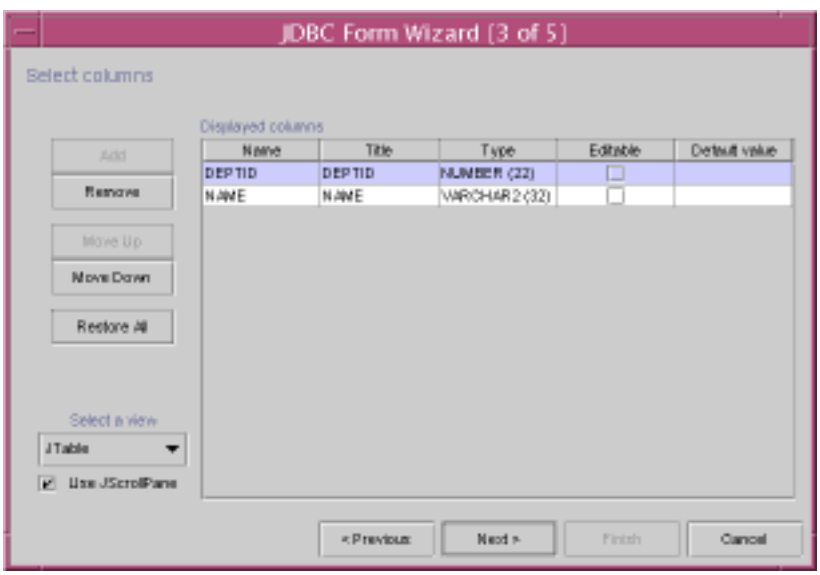


Figure 6 *JDBC Form Wizard, Panel 3*

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 Rowsets (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 a table from Tables. By default, the wizard selects the table for the secondary rowset.
3. Check Read-only if you want the corresponding rowset to be read-only.
4. Check Add `rowInserted` event handler to add a `rowInserted` event handler to the source code of 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.
5. 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`.
6. 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`".
7. Select a data column to use with a database join. Selecting this column displays a different field other than the primary column retrieved; however, it must be of the same data type as the primary column.

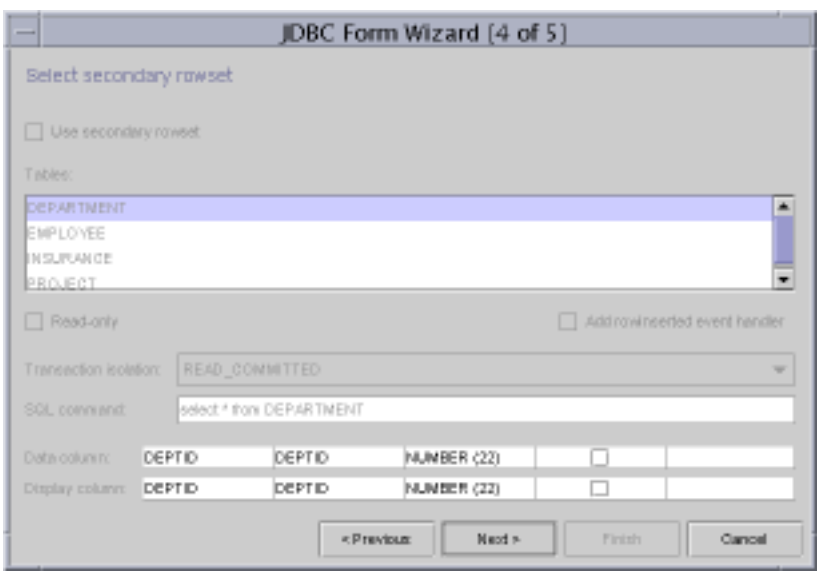


Figure 7 *JDBC Form Wizard, Panel 4*

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.

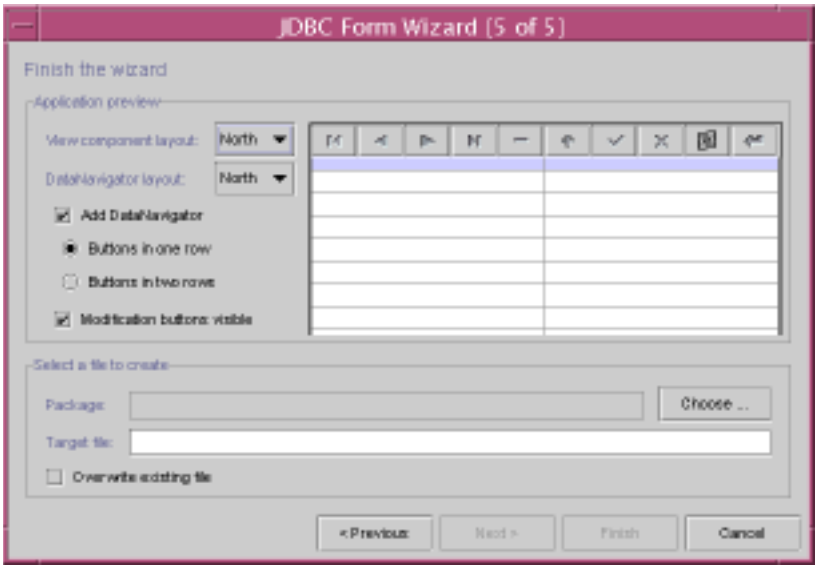


Figure 8 *JDBC Form Wizard, Panel 5*

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, JDBC, 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 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 references and 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, which returns instances of persistence-capable classes. 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

This developer creates a set of classes that models the data in a persistent data store.

➤ **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.
   - 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. Enhance the persistence-capable 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 recognize that these classes are persistence-capable and enhance them before adding to the `.jar` file. The Enhancer automatically generates all the necessary flags and method calls to mediate access to the classes' persistent fields.

Note If you choose to run the application inside Forte for Java, the enhancement will happen during the class load phase.

Developing Persistence-Aware Applications

This developer needs 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 application developer to select, update, insert, and delete data from the data store.

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 92 for more information on writing queries.

Mapped Java classes can also be accessed directly in JSP pages using Transparent Persistence tags provided as part of JSP.

System Requirements

Transparent Persistence supports development and use of persistence-capable classes with Oracle 8, PointBase, and Microsoft SQL Server.

In addition to the Transparent Persistence module, running in Forte for Java, Internet Edition, 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 7.0 driver, Version 5.1.0
- PointBase Embedded 3.4 driver with PointBase bundled in IDE
- ORACLE 8i 8.1.6 Thin

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
- `dbschema.jar` from the `modules` directory of the Forte for Java installation
- An XML SAXParser, such as `parser.jar` from `lib/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.

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 9 illustrates.

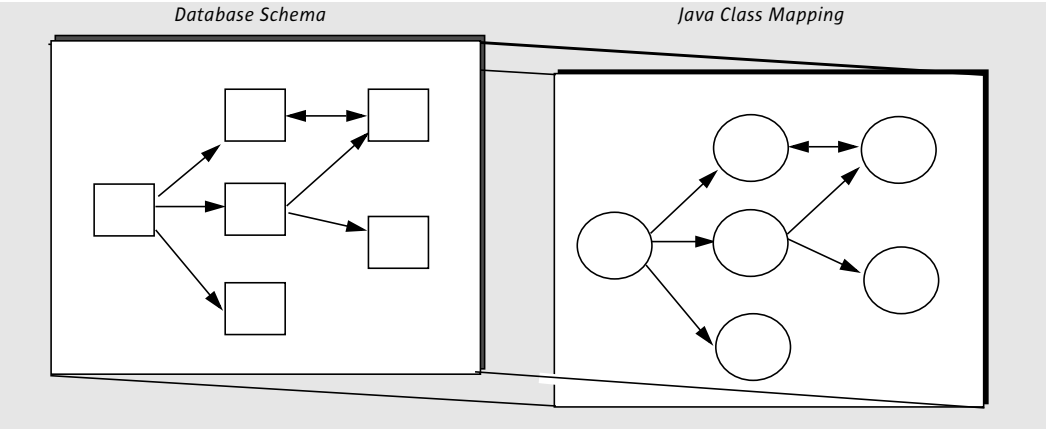


Figure 9 *Mapping a Database to Java Classes*

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->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 Map to Database 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 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 unset; you need to set them in the Properties window. See "Setting Properties" on page 62 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 Generate Java 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 10 illustrates this.

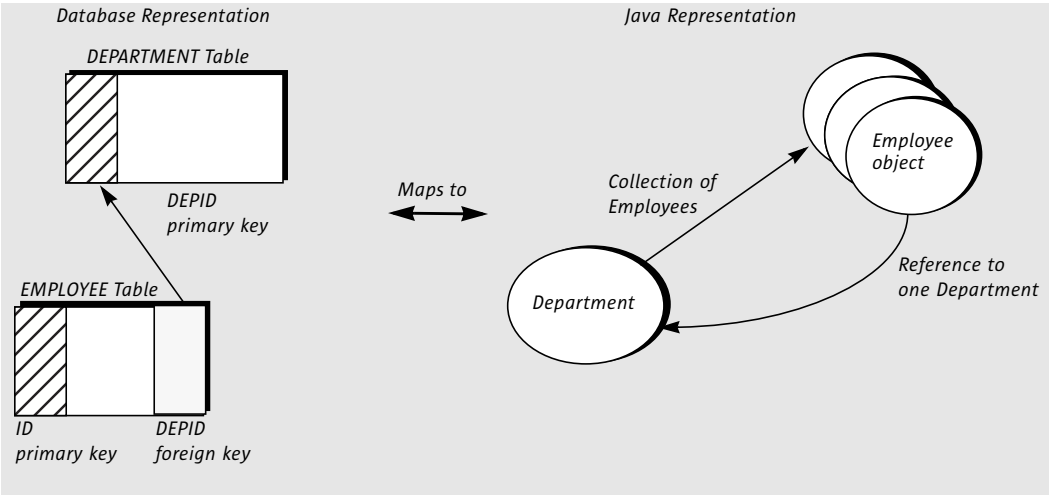


Figure 10 *Foreign Keys and One-to-Many Relationships*

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 11 shows how a many-to-many relationship might look.

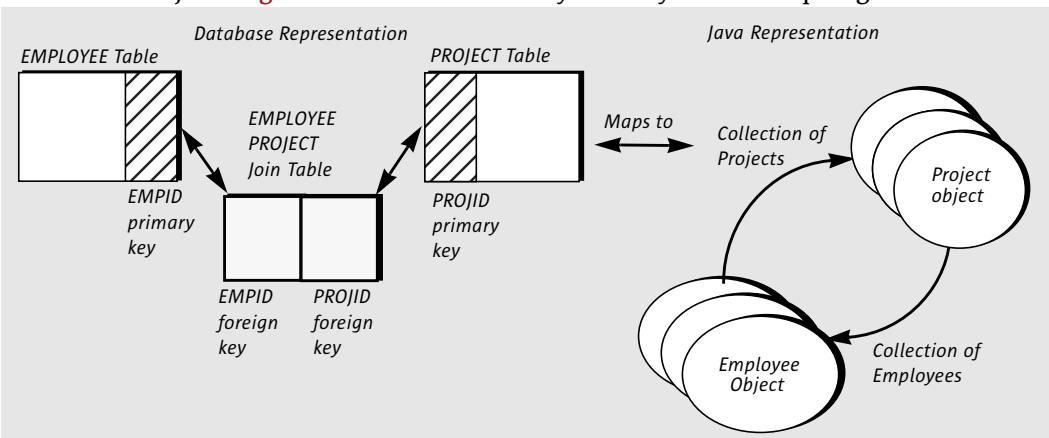


Figure 11 *Foreign Keys and Many-to-Many Relationships*

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.

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 filesystem. 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 filesystem 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 > DBSchema.
- Choose New from the File menu and then, in the Template Chooser, double-click Databases and select DBSchema.
- Select Capture Database Schema from the Tools menu.

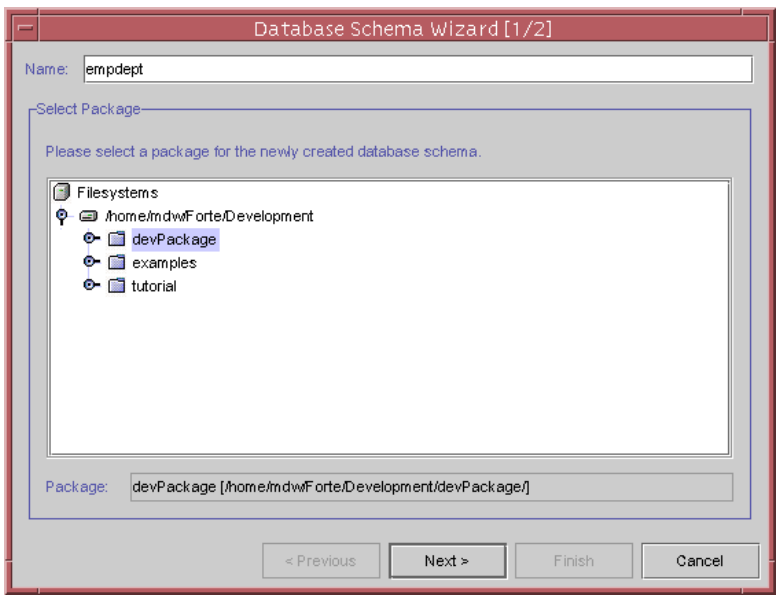


Figure 12 *Capturing a New Schema*

In the first panel, type a filename for the working copy of your schema, select a package for the captured schema, then click Next.

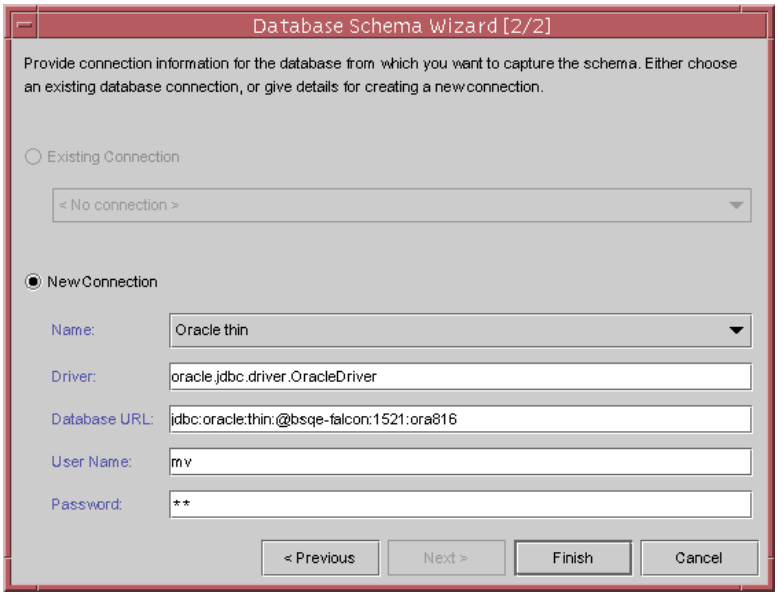


Figure 13 *Capturing a New Schema*

2 In the second panel, if you have a connection established in the Runtime tab of the Database Explorer, you can select it in the Existing Connection menu. Otherwise, under New Connection, enter the following information:

- Your system's JDBC driver.
- The JDBC URL for the database, including the driver identifier, server, port, and database name.
- 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 13 shows the Oracle driver, a server `bsqe-falcon`, and port `1521` for a database called `ora816`. Your data source will be different.
- A user name for your database.
- The password for that user.

3 Click Finish to capture the schema.

Creating Persistence-Capable Classes

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

- Database->Java mapping

  To generate Java classes from a database schema, see "Generating Persistence-Capable Classes From a Schema" on page 48.

- 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 50.

Generating Persistence-Capable Classes From a Schema

1. After you have a schema node, make sure you have a package for your persistence-capable classes (if necessary, create one by right-clicking on the filesystem and selecting New Package), select the schema node and choose the Generate Java command. This displays the Generate Java Wizard (see Figure 14), which allows you to:

   - Indicate which database tables to use for generation.

     Each table is listed under Tables 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. 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.

   - Indicate which package will contain the classes. If classes with the default names already exist in the package, you will be asked whether you want to overwrite them after you finish the wizard.

   - Indicate whether to make the generated classes persistence-capable. The default is yes. This will map the generated classes to the schema and tables.

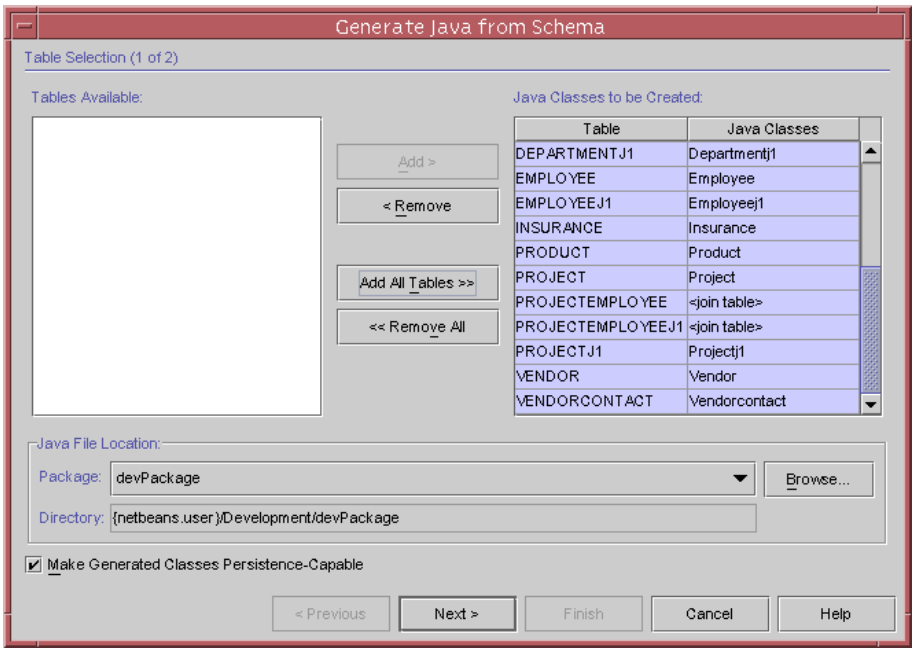


Figure 14 *Generating Java from Schema*

2 Click Next for the wizard to determine which classes will be generated, and whether relationships can be mapped properly, as shown in Figure 15.

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

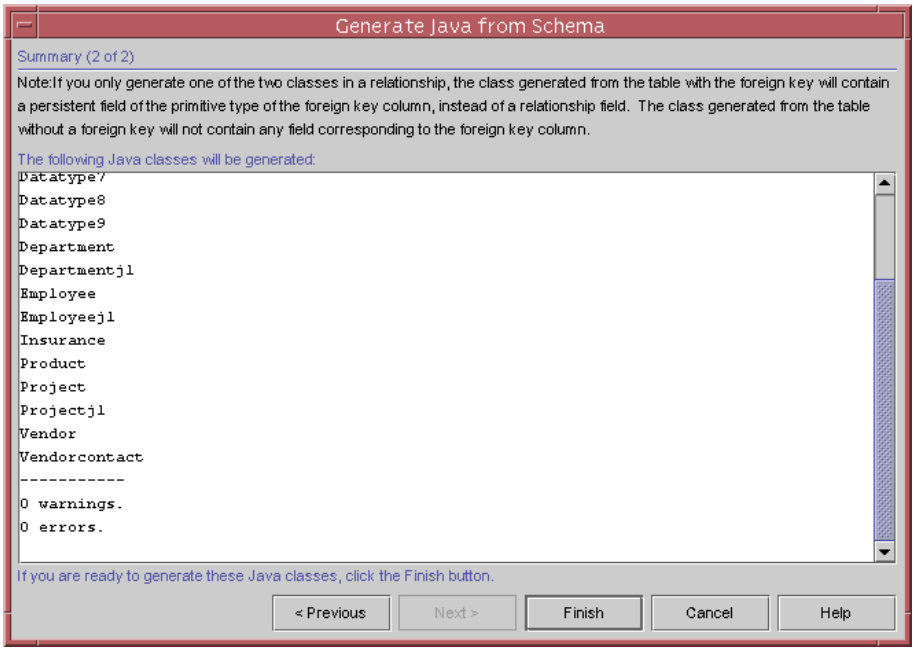


Figure 15 *Generating Java from Schema*

3 Click Finish to create a persistence-capable class for each table you selected (except for join tables), and map all fields and relationships. If you did not accept the default to create a mapping, nonpersistence-capable Java files are generated instead.

   If you want to customize your mapped classes, see "Mapping Persistence-Capable Classes" on page 51.

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 46 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 51 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 51 for instructions on how to do this.

You can edit an existing mapping by returning to the Map to Database 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 to 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 Map to Database 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 an existing class, begin by converting any relationship classes. For each class that you want to convert, right-click on the class and select Convert to Persistence-Capable.

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; nonpersistent fields are displayed with a circle.
2. In the Properties window, click on the Persistent property to activate the drop-down menu, then select True.

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

Mapping Persistence-Capable Classes

➤ **To map classes to tables using the Map to Database wizard:**

1. Right-click the class and choose the Map to Database command. This displays the Map to Database wizard (see Figure 16).

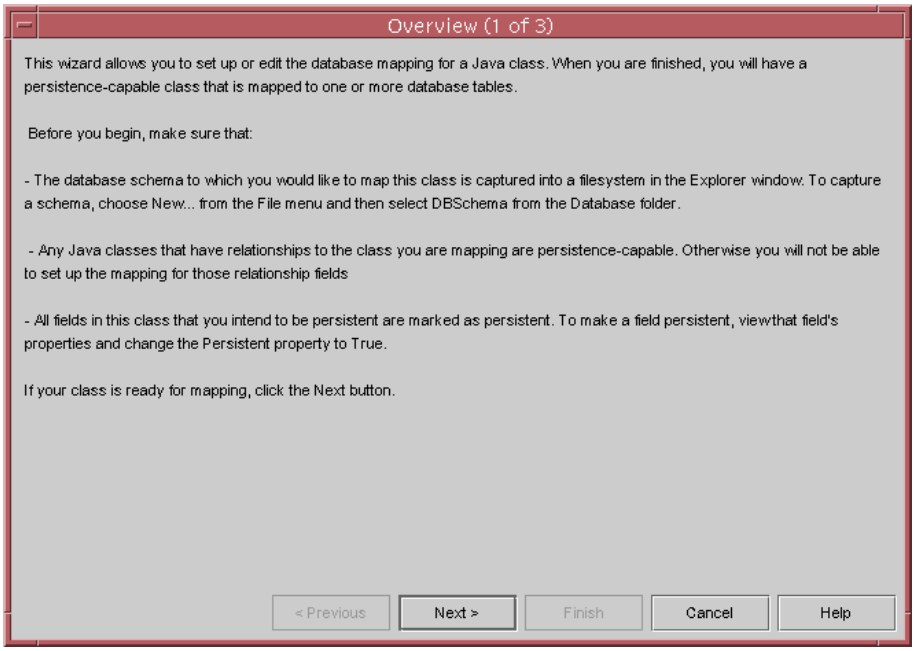


Figure 16 *Map to Database Wizard Overview Panel*

2 If you have completed the preliminary tasks, click Next to bring up the Select Tables panel of the wizard (see Figure 17). Otherwise, click Cancel, complete the tasks, and restart the wizard.

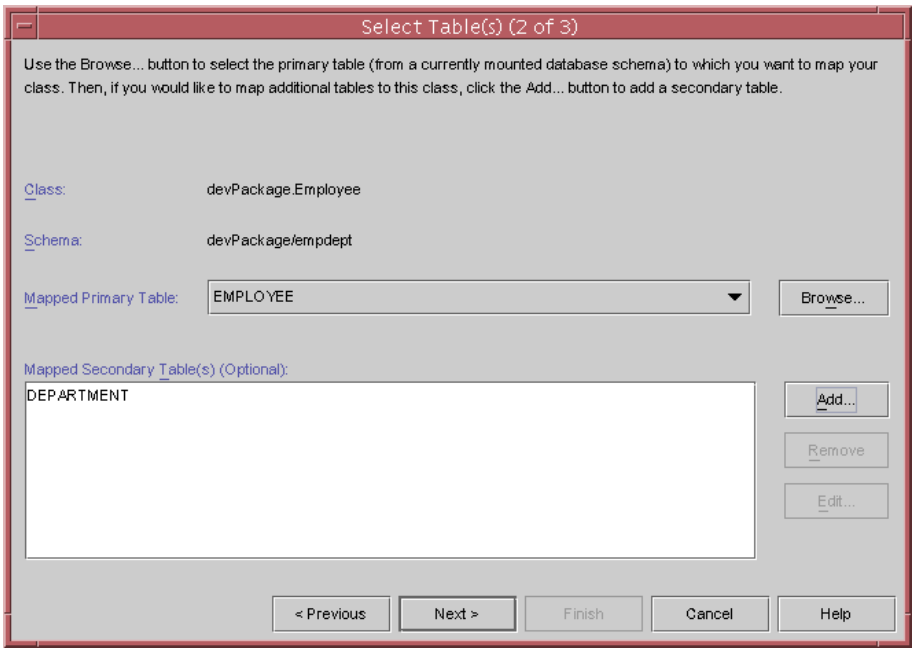


Figure 17 *Map to Database Wizard Select Tables Panel*

3 Select a primary table. Click Browse to bring up the Mapped Primary Table Editor (Figure 18).

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.

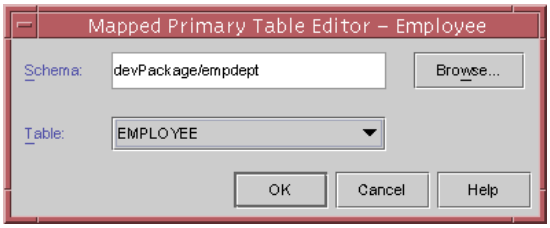


Figure 18 *Mapped Primary Table Editor*

4 In the Mapped Primary Table Editor, click Browse to select from available schemas and choose your schema in the Select Schema window (Figure 19).

Figure 19 *Select Schema Window*

After you select a schema, a menu of tables for that schema will appear in the Mapped Primary Table Editor.

5 Click OK in the Mapped Primary Table Editor to return to the Select Tables panel.

6 After the primary table is set up, you can optionally map one or more secondary tables by clicking Add to bring up the Mapped Secondary Table Setup dialog box (Figure 20).

A secondary table allows you to map your field directly 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.

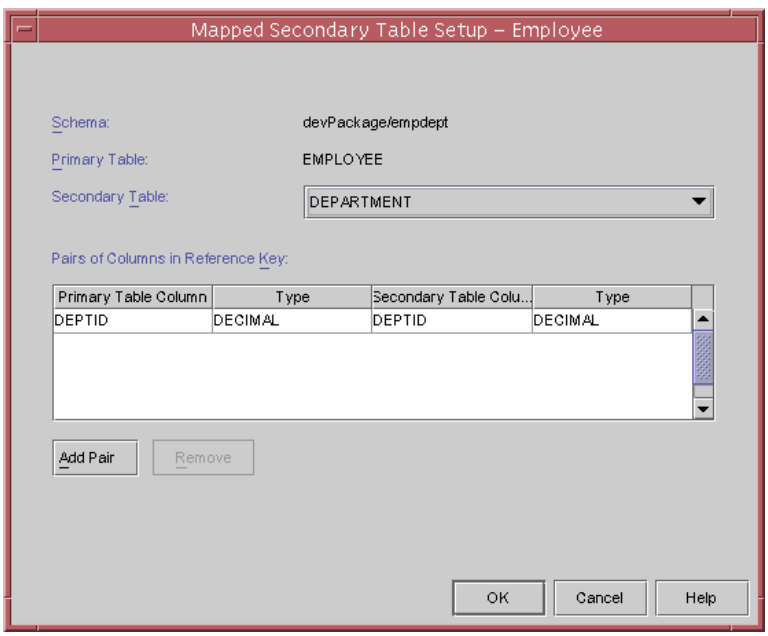


Figure 20 *Secondary Table Settings*

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 in the primary table in the database. 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.

If there is no foreign key in the database schema, the editor displays <Choose Column>, and you need to select another reference key between the two tables.

7 Click on the <Choose Column> field and select a column from the Primary Table drop-down menu. After you pick a primary table column, the choices in the secondary table column are limited to columns of compatible types. If none of the columns are compatible, the drop-down menu says <No compatible columns>. If you select a primary table column that is incompatible with your secondary table column, the value of the secondary table column will revert to <Choose Column>.

If you want to create a compound reference key, you can add and remove column pairs with the Add Pair and Remove buttons. When you click Add Pair a new pair of columns appears with the default value of <Choose Column>.

If there is no logical reference key—if no pair of columns seems to relate in a logical manner—you may want to reconsider your choice of a secondary table.

8 Click OK to save your selections.

9 Click Next in the Map to Database Wizard to bring up the Field Mappings panel of the wizard (see Figure 21).

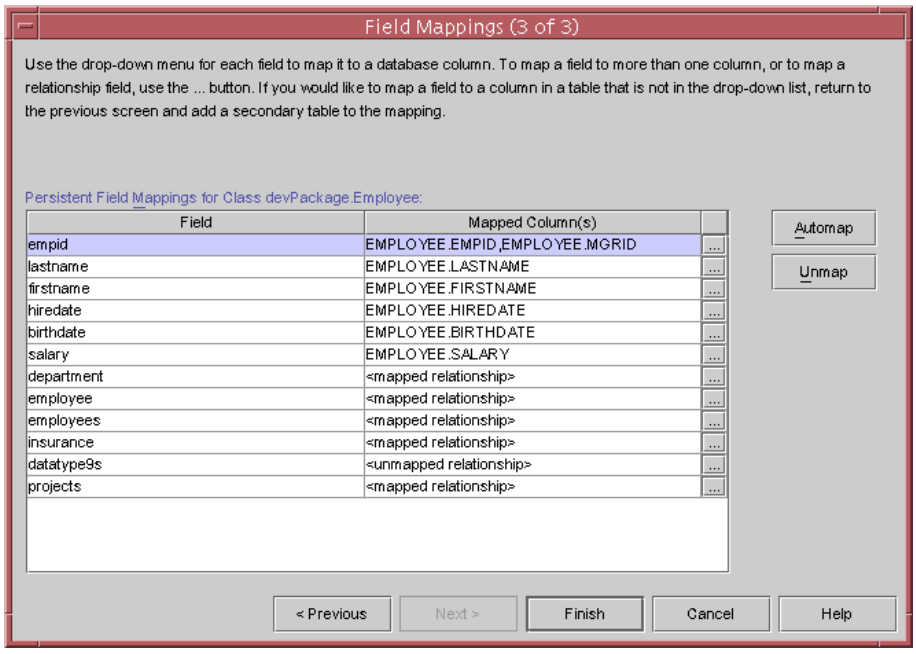


Figure 21 *Map to Database Wizard Field Mappings Panel*

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 your existing settings.

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(s) 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.

To map a field to multiple columns, see "Mapping Fields to Multiple Columns" on page 57.

To map a relationship field, see "Mapping Relationship Fields" on page 57.

Mapping Fields to Multiple Columns

➤ **To map a field to multiple columns:**

1 Click "..." in the Field Mappings panel to display the Map Field to Multiple Columns dialog box (see Figure 22).

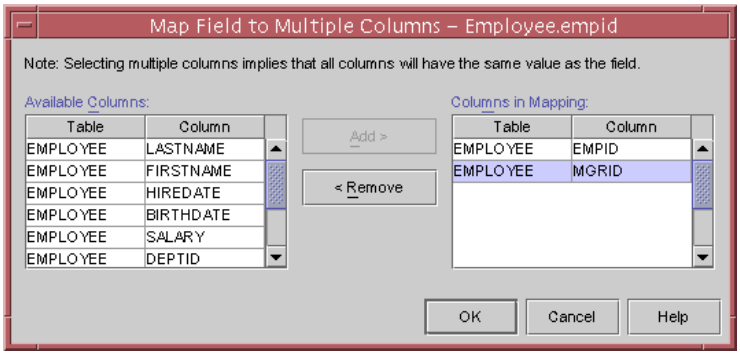


Figure 22 *Map Field to Multiple Columns Dialog Box*

2 Select one or more columns from the left side of the dialog box and use the Add and Remove buttons to specify the columns to be mapped.

Columns are from the tables that you have mapped. If you do not see the column that you want to map, you might need to add a secondary table to your mapping, or change the primary table you 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 same value. Transparent Persistence writes the same field value to all columns, in the order the columns are listed; it reads the value from the first column listed. Therefore, if the value of one of the columns is changed, it will only be read if the change was made to that first column. Otherwise, the original value will be read. Writing a value to the database would overwrite any conflicting changes made to any other columns.

3 Click OK to save the mapping.

Mapping Relationship Fields

When you have foreign key relationships 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.

➤ **To map a relationship field:**

1 Click on "..." in the Field Mappings panel next to the drop-down menu of a relationship field to bring up the Map Relationship Field dialog box (Figure 23).

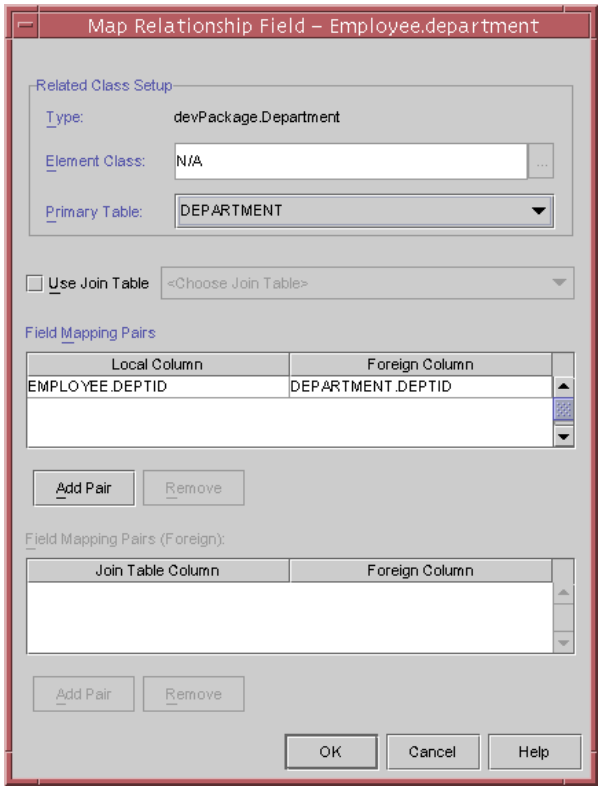


Figure 23 *Relationship Dialog Box*

The Related Class Setup portion of the dialog box has three fields: Type, Element Class, and Primary Table.

The Type field shows the type of relationship field. In a many-to-one or many-to-many relationship, the type is the Collection type (Figure 24).

The Element Class field is only active for one-to-many or many-to-many relationships. (For one-to-one relationships, the field displays `N/A`). The Element Class is the type of the object in the Collection.

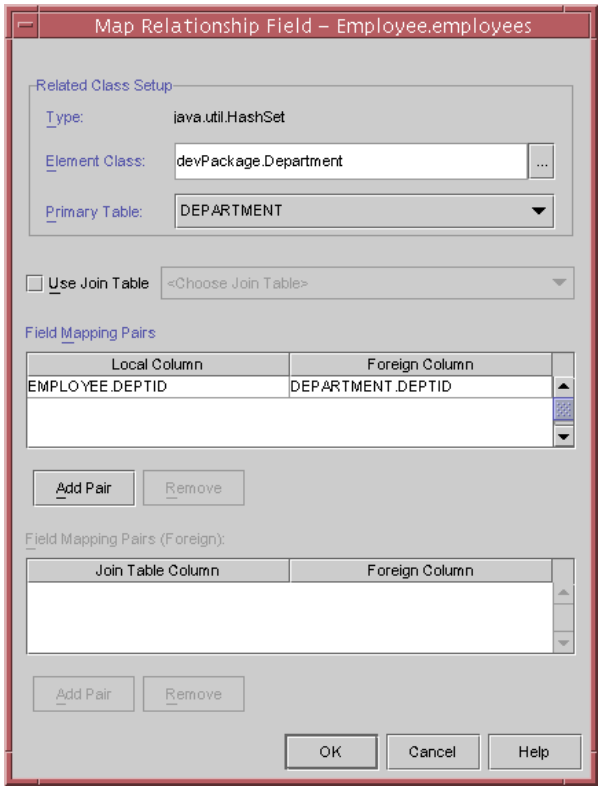


Figure 24 *Relationship Dialog Box for a Collection*

If the Element Class is unselected, the field will display <Choose Element Class>. Select the Element Class by clicking the "..." button next to the Element Class field. This displays a file chooser that lets you select a persistence-capable class from the Filesystems tree (Figure 25).

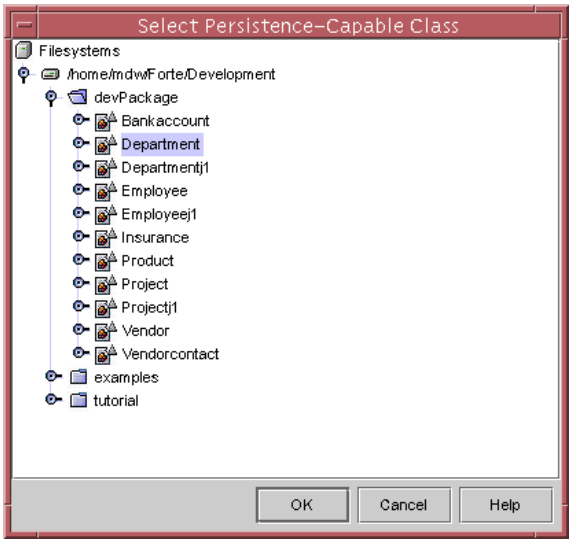


Figure 25 *Select Persistence-Capable Class Window*

The Element Class can also be selected in the Properties window. See "Setting Properties" on page 62 for more information.

If Transparent Persistence can determine the primary table, the table name is displayed; otherwise, select the table from the Primary Table drop-down menu.

In many cases, Transparent Persistence can determine the information in the Field Mapping Pairs section automatically. By default, Local Column displays the foreign key of the table you are mapping and Foreign Column displays the primary key of the table to which it refers.

If there is no foreign key relationship, <Choose Column> is displayed under Local Column and Foreign Column. Click on each <Choose Column> entry and select a column from the drop-down menu. You can also use the drop-down menu to change an existing setting.

Use the Add Pairs button to create additional field mapping pairs.

If your relationship is many-to-many, click on Use Join Table to select a join table from the drop-down menu (Figure 26).

The Field Mapping Pairs section changes to Field Mapping Pairs (Local) and the Field Mapping Pairs (Foreign) section becomes active. By default, the Local Column is the primary key of the table you are mapping and the Foreign Column is the primary key of the related table. The Join Table Columns are the foreign keys referring to each of the tables.

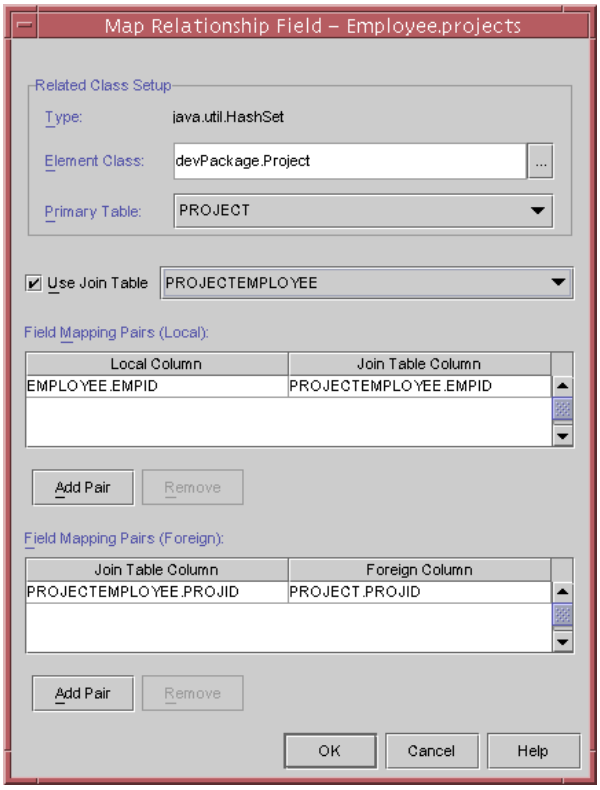


Figure 26 *Relationship Dialog Box Using a Join Table*

Setting Properties

Persistence-capable classes and persistent fields have several unique properties that can be specified outside of the Mapping wizard. Table 7 describes the properties unique to persistence-capable classes.

Table 7 *Properties for Persistence-Capable Classes*

Property	Description
Key Class	An associated class that includes a key field that uniquely identifies a persistence-capable instance.
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 50 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 directly; you must capture the schema as described in "Capturing a Schema" on page 46.
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.
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 51.

Figure 27 shows the properties for a persistence-capable class.

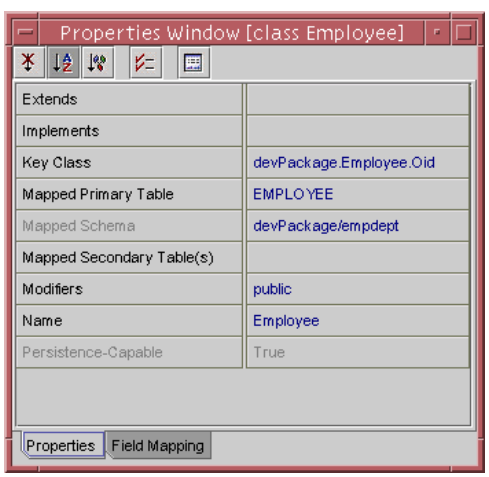


Figure 27 *Properties for a Persistence-Capable Class*

The Key Class is set to `devPackage.Employee.Oid`. This is the Oid class set by the Java generator automatically. If you use meet-in-the-middle mapping, you must set the Key Class manually. See "Key Fields and Key Classes" on page 66 for more information on setting the Key Class.

You can unmap a class by clicking "..." on Mapped Primary Table, then choosing `<unmapped>` from the drop-down menu. 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.

Unmapping a class can cause you to lose any customizations you have made to the mapping of a class for this table. Even if you decide to map the table in a subsequent operation, prior customizations will not be reinstated. The same holds true for other unmapping operations. For example, you will also lose customizations made to a field mapping if you unmap the field.

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 28). These properties are only available once you have mapped a primary table.

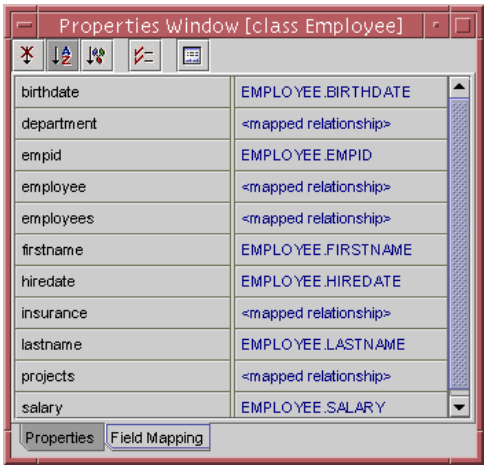


Figure 28 *Properties for a Persistence-Capable Class*

Map a persistent field by choosing a column from the field's drop-down menu. To map additional columns to that field, click "..." to display the Map Field to Multiple Columns dialog box. See "Mapping Fields to Multiple Columns" on page 57 for an explanation of the dialog box.

Map a relationship field by selecting the field and clicking "..." to display the Map Relationship Field dialog box. See "Mapping Relationship Fields" on page 57 for an explanation of the dialog box.

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

Table 8 describes the properties unique to persistent fields.

Table 8 *Properties for Persistent Fields*

Property	Description
Delete Action (Relationship fields only)	This is either Cascade or None. Cascade means that when this field is deleted, all related fields will be deleted with it. None means that only the specified field will be deleted.
Element Class (Relationship fields only)	If the type of a relationship field is a collection type, you must specify an Element Class for the field. The Element Class is the type of persistence-capable object that makes up the collection.
Fetch Group	Transparent Persistence has two fetch group settings: default and none. A setting of default for a field means that field will be fetched along with all other fields that have a setting of default. By default, the default fetch group includes all persistent fields except relationship fields, which must have a setting of none. If the Fetch Group property is greyed out, the field is either unmapped or a relationship field.
Key Field	Whether the field is mapped to the primary key of the persistence-capable class's primary table. The value is either True or False.
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.
Persistent	Whether the field is persistent or not. The value is either True or False.
Read Only	Whether the field is read-only or not. The value is either True or False.
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 * (unlimited). On the one side of a relationship, the Upper Bound is 1 and cannot be changed.

Figure 29 shows a Properties window for an mapped persistent field called `empid`, in the `Employee` class.

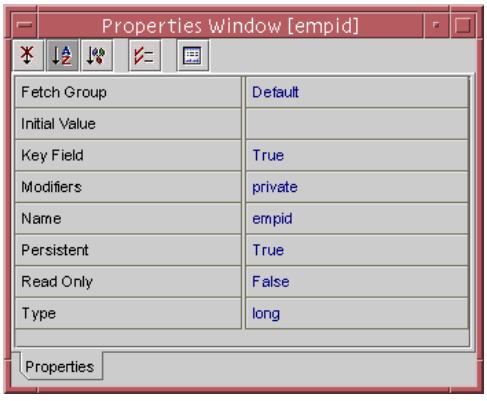


Figure 29 *Properties for a Persistent Field*

The Key Field is set to True, because `empid` is mapped to the primary key of `Employee's` primary table. See "Key Fields and Key Classes" on page 66 for more information on setting the Key Field.

Figure 30 shows a Properties window for a persistent relationship field.

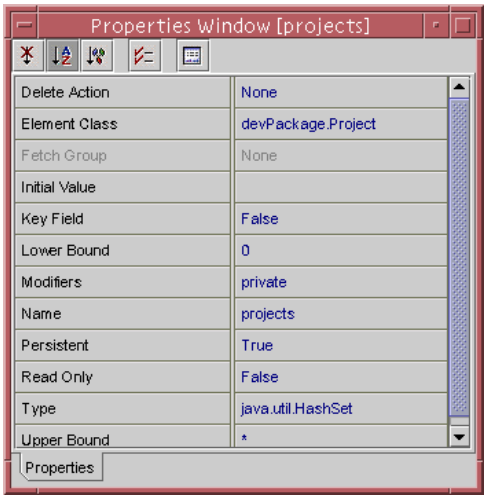


Figure 30 *Properties for a Persistent Relationship Field*

The Delete Action is set to None, so deleting this field would not delete any related fields. The Fetch Group property is greyed out because relationship fields cannot be part of the default fetch group.

The field in Figure 30 is a collection, so the Type is a collection type. By default, the Lower Bound and Upper Bound for collections are 0 and *, respectively, meaning that the collection can contain any number of elements, and can also be null. Non-collection relationship fields have an Upper Bound of 1.

The field icons in the Explorer change to indicate different class and field types, as shown in Figure 31 and Figure 32.

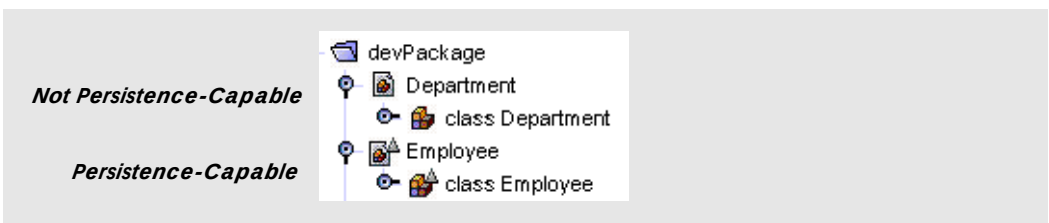


Figure 31 *Class Icons*

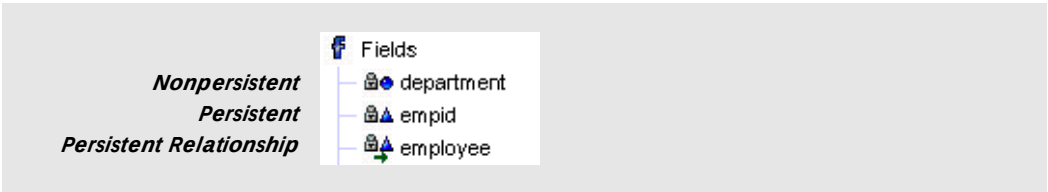


Figure 32 *Field Icons*

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.

A Key Class can be either of the following types:

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

Both suffixes are case-insensitive.

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

Following is an example of a Key Class defined for the `Employee` class. It matches the settings in Figure 27 and Figure 29. Figure 27 shows the Key Class set to `devPackage.Employee.Oid` and Figure 29 shows the `empid` field set to true. Since `empid` is the only field in the Key Class, all other fields in the `Employee` class would have a Key Field setting of False.

```
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 );
}
```

Enhancing

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.

- Create a `.jar` file with Tools > Add to Jar

  Use this approach when you are planning to run your application outside of Forte for Java. Do not add the Java files, as this can result in unexpected `javac` errors in future compilations. Also, make sure that your schema file is included. If the schema file is in the specified package, it will be included automatically; otherwise, you will need to specify its location.

- Run the application in Forte for Java

  Select the class that contains your application's `main()` method and select Persistence Executor in the Properties window.

In each of these cases, the Transparent Persistence Enhancer runs automatically 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.

The enhancement process can be repeated by using Add to Jar with another `.jar` file. If a set of nonenhanced persistence-capable classes is moved to another environment, they can be enhanced in that environment, which will generate the appropriate code.

Note Do not place persistence-capable classes in the Classes directory of the web module. Class enhancement occurs only in two cases: when you use Forte for Java's Persistence Executor and when you use the JAR Packager. A web module cannot use the Persistence Executor, and does not use the plain JAR Packager. Instead, it uses its own version (the WAR packager), which does not enhance persistence-capable classes. Therefore, you must package persistence-capable classes into a JAR (using the plain JAR Packager) before placing them in the `lib` directory of the web module.

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>`.

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 9 lists these data types and notes whether each type is supported.

Table 9 *Supported Data Types*

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

Table 10 lists the nullability of supported data types.

Table 10 *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`
`java.lang.Double`	`DOUBLE`	`NULL`

Table 10 *Data Type Conversions in Mappings (Continued)*

Java Type	JDBC Type	Nullability
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
java.lang.String	VARCHAR	NULL
java.lang.String	LONGVARCHAR	NON NULL
java.lang.String	LONGVARCHAR	NULL
java.util.Date	DATE	NON NULL
java.util.Date	DATE	NULL
java.util.Date	TIMESTAMP	NON NULL
java.util.Date	TIMESTAMP	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 33 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.

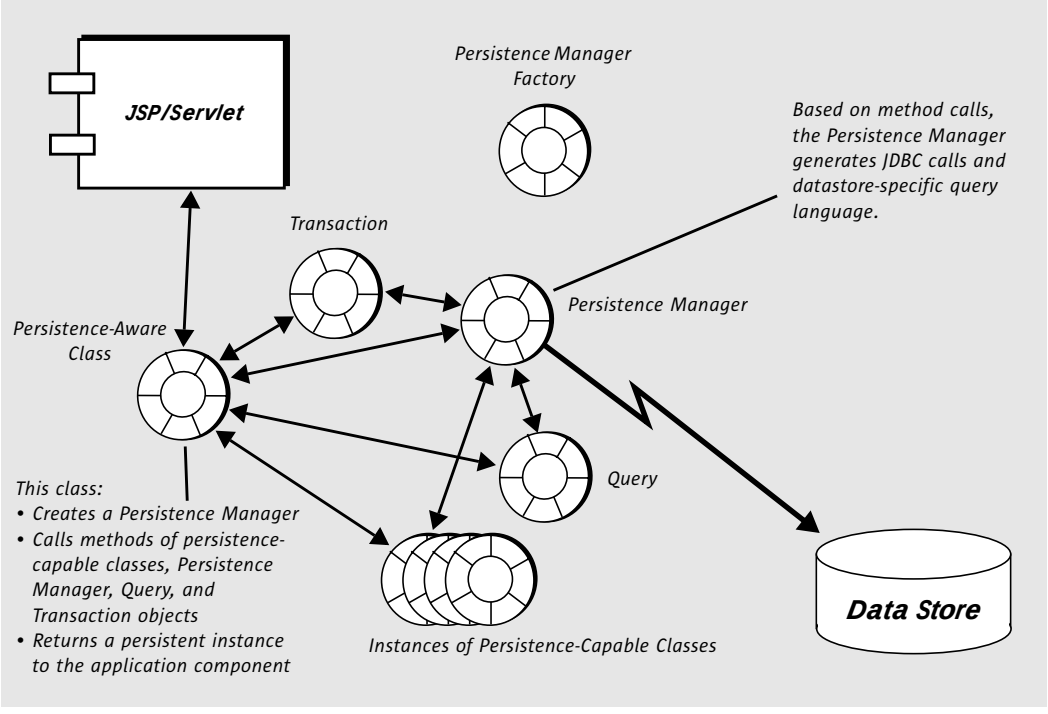


Figure 33 *Moving Persistence-Aware Logic to Its Own Class*

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 a `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 33, 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.

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 77 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 80 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 80 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 83 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 34 presents these steps in a flowchart.

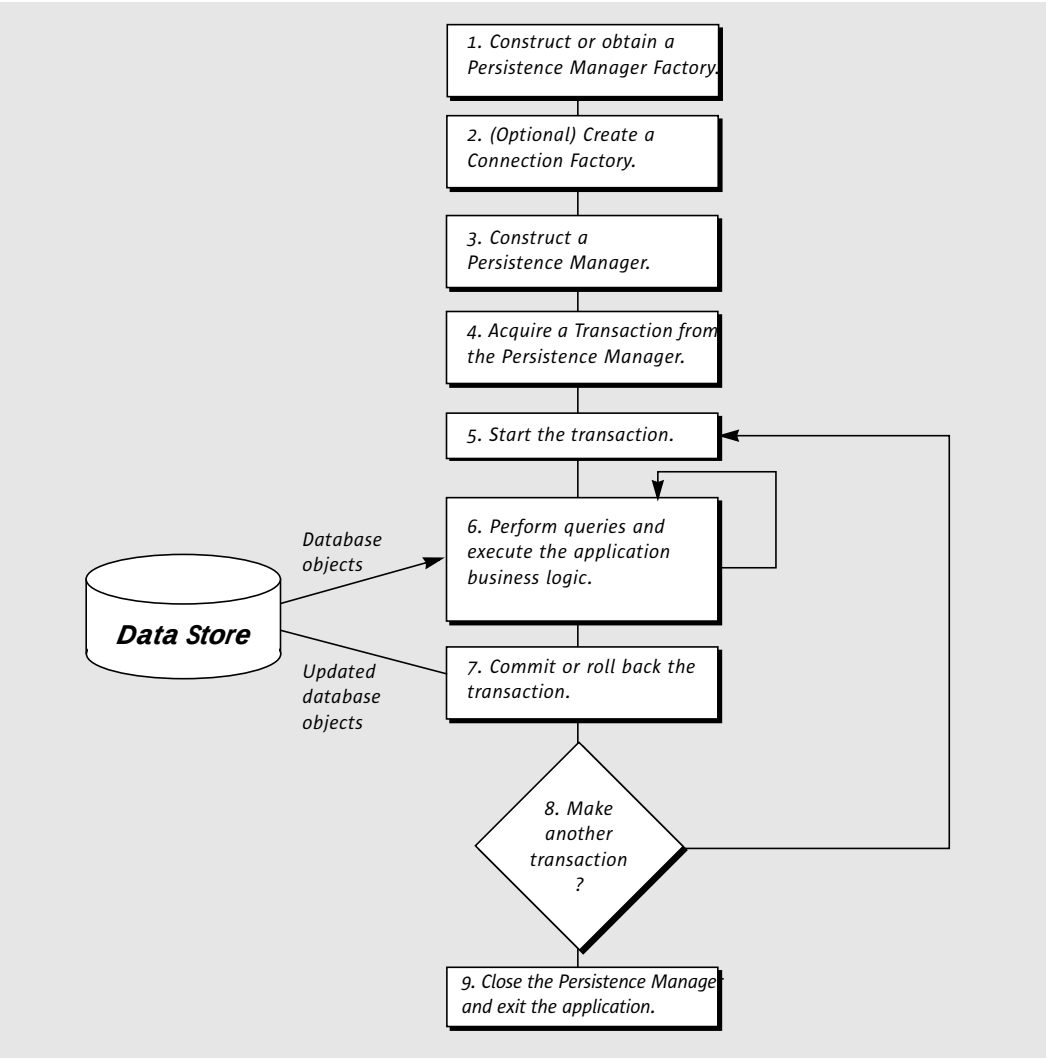


Figure 34 *Transparent Persistence Application Logic*

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 80.

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 11 discusses each method in detail.

Table 11 *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 11 *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.

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 nonmanaged environment, 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 12 discusses each method in detail.

Table 12 *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.
`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 13 discusses each method in detail.

Table 13 *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.
`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.

Table 13 *PersistenceManager Methods (Continued)*

Method	Description
`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))`
`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.
`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();
try {
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 14 discusses each method in detail.

Table 14 *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.

Table 14 *Transaction Methods (Continued)*

Method	Description
`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.
`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 nonmanaged 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`.

`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.

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 15 shows which access inconsistencies are possible under each of these settings.

Table 15 *Isolation Levels*

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

Table 15 *Isolation Levels (Continued)*

Level	Dirty Read	Nonrepeatable Read	Phantom Insert
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 85 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. If `retainValues()` is set to true (the default), the state of the instance will be reloaded automatically. If `retainValues()` is set to false, the state of the instance will be reloaded when the instance is modified.

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 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 01 and wanted to update 02, and the other transaction had read 02 and wanted to update 01. The outcome of this deadlock depends on which DBMS you are using.

Microsopft 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 are left untouched.

  Note — Transparent Persistence does not automatically nullify relationships. Be sure to explicitly nullify the relationship before deleting the instance, or you might get a constraint violation from the database.

- 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 Properties" on page 62.

Warning 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 16.

Table 16 *Query Elements*

Element	Requirement	Description
Result class	Required	All objects of the result collection are of this class. The class is used to scope the names in the query filter. The result class of a query must be persistence-capable. It is defined by a `newQuery` argument or by the Query method `setClass`.

Table 16 *Query Elements (Continued)*

Element	Requirement	Description
Candidate collection	Required	This is the extent collection (see the `PersistenceManager.getExtent` method) of the result 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 result 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 result 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 result class, candidate collection, and filter specified. Other options are shown in Table 17.

Table 17 *newQuery Options*

Method	Description
`Query newQuery()`	Construct an empty query instance.

Table 17 *newQuery Options (Continued)*

Method	Description
`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 result class specified.
`Query newQuery (Class cls, Collection cln)`	Construct a query instance with the result class and candidate collection specified.
`Query newQuery (Class cls, String filter)`	Construct a query instance with the result class and filter specified.
`Query newQuery (Class cls, Collection cln, String filter)`	Construct a query instance with the result class, the candidate collection, and filter specified.

Table 18 discusses each method of the `Query` interface in detail.

Table 18 *Query Interface Methods*

Method	Description
`void setClass (Class resultClass)`	Binds the result 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 statements to 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 statements 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.

Table 18 *Query Interface Methods (Continued)*

Method	Description
`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 result 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 Michael and a last name of Bouschen. The second line filters elements with a first name of Michael or a salary greater than 200,000.

```
q.setFilter("firstname == \"Michael\" & lastname == \"Bouschen\"");
q.setFilter("firstname == 'Michael' | salary > 200000.0");
```

Identifiers in the filter expression denote fields of the result class, unless the name is defined as a parameter 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 == \"Michael\"");
q.setFilter("this.firstname == \"Michael\"");
```

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`, `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("Michael");
```

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("Michael")` returns a collection of `Employee` instances with a `firstname` value of `Michael`. 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.

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 relationships 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"`. However, you cannot compare a relationship field with null. `setFilter("department == null")` will result in a `JDOUnsupportedOptionException`.

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.

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 19:

Table 19 *Query Operators*

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

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.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.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.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.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 DEPTARTMENT.DEPTNAME = 'Engineering' 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.Employee");
Collection clnEmployee = pm.getExtent (empClass, false);
String filter = "dept.name == Engineering";
String param = "String Engineering";
Query q = pm.newQuery (empClass, clnEmployee, filter);
q.declareParameters (param);
String rnd = "Engineering";
Collection emps = (Collection) q.execute (rnd);
```

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);
```

Warning Once the `Lineitem.ordernumber` is set to `1`, it can only be added to Order `1`'s `lineitems` collection. The runtime does not verify this.

```
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

```
o.getLineitems().remove(lt);
pm.deletePersistent(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 can corrupt `Order` or cause a constraint violation in the database.

```
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 be 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.

Transparent Persistence has two fetch group settings: default and none. A setting of default for a field means that field will be fetched along with all other fields that have a setting of default.

By default, Transparent Persistence includes all persistent fields except relationship fields in the default fetch group. 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 the 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 (generated during the enhancement process), that includes a field typed for a value that uniquely identifies 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.

The nature of each identity field for a class is determined during enhancement. 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 inner class with the suffix `Oid` (default)
- Separate class with suffix `Key`

Both suffixes are case-insensitive.

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 (database or primary key), 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, mapping generates a public static inner 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.

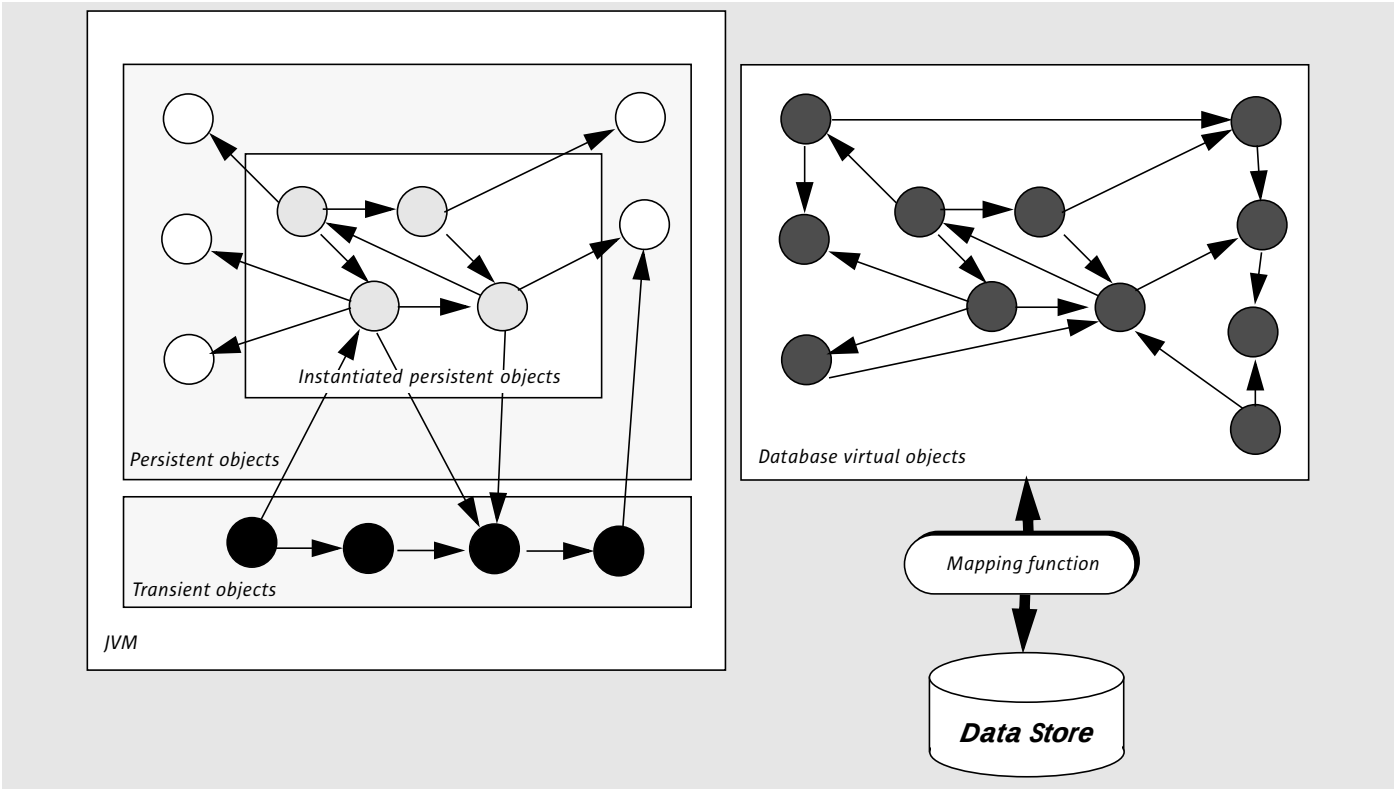


Figure 35 *Instantiated Persistent Objects*

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 nonpersistent 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 nonpersistent, or nontransactional nonpersistent.

Persistent Fields

Table 20 describes the persistent field types.

Table 20 *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 Transient Fields

A persistence-capable class can have both persistent fields and transient 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.
- Transient 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 21 summarizes the exceptions associated with the rule violations.

Table 21 *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.
`JDOQueryException` (“Unbound query parameter”)	There is a query syntax error.
`JDOFatalUserException` (“PM is closed”)	You cannot access a closed Persistence Manager.
`JDOFatalInternalException`	There has been an unexpected error at mapping or runtime.

Table 21 *JDO User Exceptions (Continued)*

Exception	Explanation
`JDOUnsupportedOptionException`	You cannot use an unsupported option (for example, `setIgnoreCache(false)`).
`JDODataStoreException`	There is a conflict in the database or an integrity constraint vioation.
`JDOQueryException` ("Missing result class specification.")	The result 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 result class <class>.")	The candidate collection is not the extent collection from the result 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.
`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 `JDOQueryExecption("setFilter column(6): Field 'michael' not defined for class 'com.xyz.Employee'.")`, if the class `Employee` does not define a field `michael`.

Appendix A

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 Persistence Manager 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" />
```

Index

I

J

K

M

N

O

P

Q

R

S

T

U

V

W