Oracle9*i*AS TopLink

Mapping Workbench Reference Guide

Release 2 (9.0.3)

August 2002

Part No. B10063-01

Oracle9*i*AS TopLink Mapping Workbench Reference Guide, Release 2 (9.0.3)

Part No. B10063-01

Contents

1 Understanding the Workbench

2 Understanding Projects

3 Understanding Databases

4 Understanding Descriptors

A Object Model Requirements

Index

Send Us Your Comments

Oracle9*i*AS TopLink Mapping Workbench Reference Guide, Release 2 (9.0.3)

Part No. B10063-01

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

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

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

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

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

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

Preface

This reference guide includes the concepts required for using the TopLink Mapping Workbench, a stand-alone application that creates and manages your descriptors and mappings for a project. This document includes information on each Mapping Workbench function and option.

This preface contains the following topics:

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

Intended Audience

This document is intended for TopLink users who are familiar with the object-oriented programming and Java development environments.

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

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

Documentation Accessibility

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

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

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

Structure

This document includes the following chapters:

Chapter 1, "Understanding the Workbench"

This chapter introduces the Mapping Workbench – a tool to graphically configure descriptors and map your TopLink project.

Chapter 2, "Understanding Projects"

This chapter contains instructions for creating and maintaining TopLink project files, including workbench preferences and team development.

Chapter 3, "Understanding Databases"

This chapter describes how to create database sessions and import/export database tables to and from your TopLink project.

Chapter 4, "Understanding Descriptors"

This chapter summarizes TopLink descriptors, including standard and advanced properties and mappings.

Chapter 5, "Understanding Direct Mappings"

This chapter summarizes the direct mapping types supported by TopLink.

Chapter 6, "Understanding Relationship Mappings"

This chapter summarizes the relational mapping types supported by TopLink.

Chapter 7, "Understanding Object Relational Mappings"

This chapter summarizes the object relational mapping types supported by TopLink.

Appendix A, "Object Model Requirements"

This section summarizes TopLink's object model requirements.

Related Documents

For more information, see these Oracle resources:

Oracle9*i*AS TopLink Getting Started

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

Oracle9*i*AS TopLink Tutorial

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

Oracle9*i*AS TopLink Foundation Library Guide

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

Oracle9*i*AS TopLink Mapping Workbench Reference Guide

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

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

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

Oracle9*i*AS TopLink Troubleshooting

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

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

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

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

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

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

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

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

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

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

Conventions

Conventions in Text

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

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id` and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true.` Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

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

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

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

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

. . .	Horizontal ellipsis points indicate either:
	■ That we have omitted parts of the code that are not directly related to the example
	■ That you can repeat a portion of the code
		CREATE TABLE ... AS *subquery*;
		SELECT *col1*, *col2*, ... , *coln* FROM employees;

. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	

Other notation	You must enter symbols other than brackets, vertical bars, and ellipsis points as shown.	acctbal NUMBER(11,2);
		acct CONSTANT NUMBER(4) := 3;

Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	CONNECT SYSTEM/*system_password* DB_NAME = *database_name*

Conventions for Microsoft Windows Operating Systems

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

Convention	Meaning	Example
Choose **Start** >	How to start a program.	To start the Oracle Database Configuration Assistant, choose **Start** > **Programs** > **...** .

Convention	Meaning	Example
Case sensitivity and file and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	`c:\winnt"\"system32` is the same as `C:\WINNT\SYSTEM32`
	IMPORTANT NOTE: File names and directory names *are* case sensitive under UNIX. Where the name of a file or directory is mentioned and the operating system is a non-Windows platform, you must enter the names exactly as they appear unless instructed otherwise.	
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`

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

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

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

1

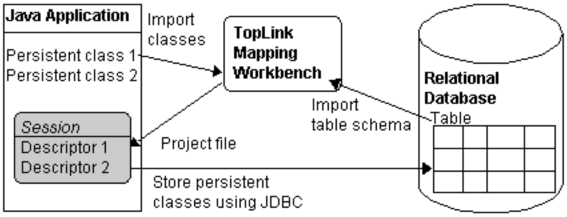
Understanding the Workbench

The Mapping Workbench is separate from Oracle 9*i*AS TopLink – it allows you to graphically configure descriptors and map your project. The Mapping Workbench can verify the descriptor options, access the data source, and create the database schema. With the Mapping Workbench you can define TopLink descriptors *without using code*.

To Use the Mapping Workbench in a Java Application:

1. Define an object model (a set of Java classes) to describe and solve your problem domain.
2. Use the Mapping Workbench to create a project, import your Java classes and relational tables, and specify descriptors to describe how the classes map to your relational model.
3. In your Java application, create a TopLink session and register your descriptors. Add logic to your application to use the session to retrieve/store objects from/to the database.

Figure 1–1 Using the Mapping Workbench



Starting the Mapping Workbench

Use this procedure to start the Mapping Workbench.

To Start the Mapping Workbench:

For Windows environments: From the **Start** menu, select **Program Files > Oracle9iAS TopLink > Mapping Workbench**.

or

For non-Windows environments: Execute the `<INSTALL_DIR>toplink\workbench\workbench.sh` file.

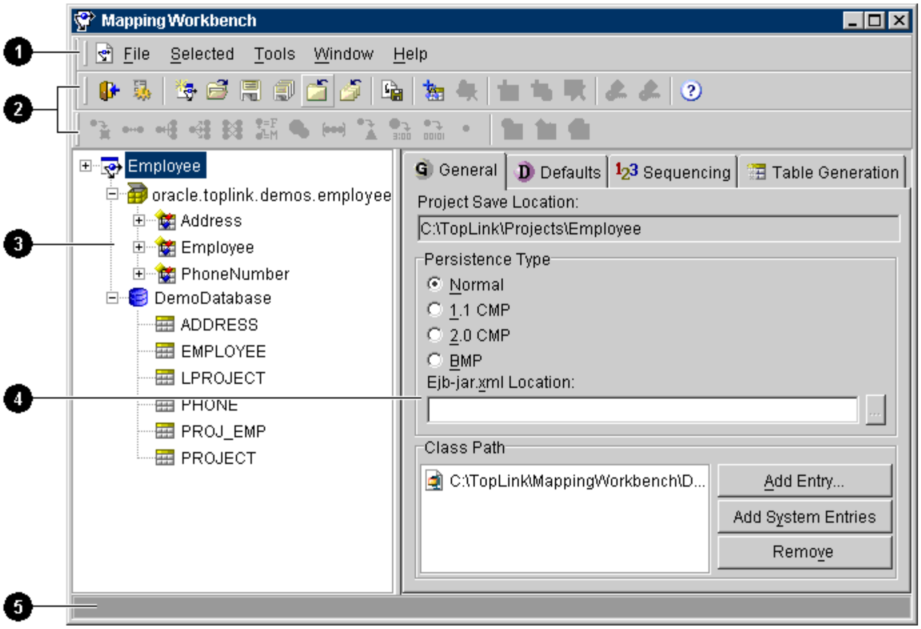
The splash screen appears, followed by the workbench window.

Working with the Workbench

The Mapping Workbench interface includes these parts:

- Menu – Pull-down menus for each Mapping Workbench function. Some objects also contain context-sensitive pop-up menus. See "Using the Menus" on page 1-3 for more information.
- Toolbar – Shortcuts to specific functions. See "Using the Toolbars" on page 1-4 for more information.
- Project Tree pane – The project tree for all open projects (see "Using the Project Tree Pane" on page 1-5). Click on the plus or minus (**+/–**) next to an object (or double-click the object) to expand/collapse the tree. When you select an object in the **Project Tree** pane, its properties appear in the **Properties** pane.
- Properties pane – Specific property tabs for the currently selected object. See "Using the Properties Pane" on page 1-7 for more information.
- Status bar – Provides instant information regarding the status of descriptors and mappings.

Figure 1–2 Mapping Workbench



Project Save Location: C:\TopLink\Projects\Employee

Persistence Type

● Normal

○ 1.1 CMP

○ 2.0 CMP

○ BMP

Ejb-jar.xml Location:

Class Path

C:\TopLink\MappingWorkbench\D...

User-interface components called out in Figure 1–2:

1. Menu bar
2. Toolbars
3. Project Tree pane
4. Properties pane
5. Status bar

Using the Menus

The TopLink Mapping Workbench contains two types of menus:

- Menu Bar Menus
- Pop-up Menus

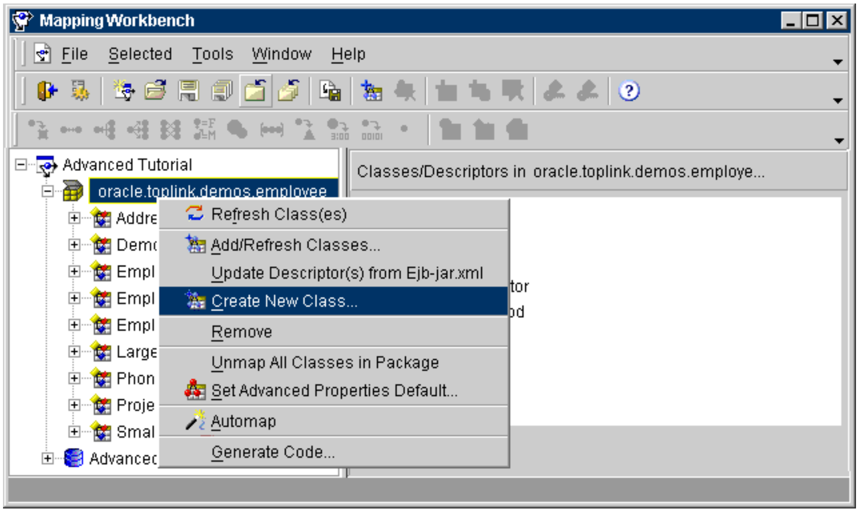
Menu Bar Menus

The menu bar provides pull-down menus for each TopLink Mapping Workbench function. Some menus (such as **Selected**) are context-sensitive – the available options may vary, depending on the currently selected object.

Pop-up Menus

When you right-click on objects in the **Project Tree** pane, a pop-up menu appears with functions specific to the selected object.

Figure 1–3 Sample Pop-up Menu



Using the Toolbars

The Mapping Workbench contains two types of toolbars:

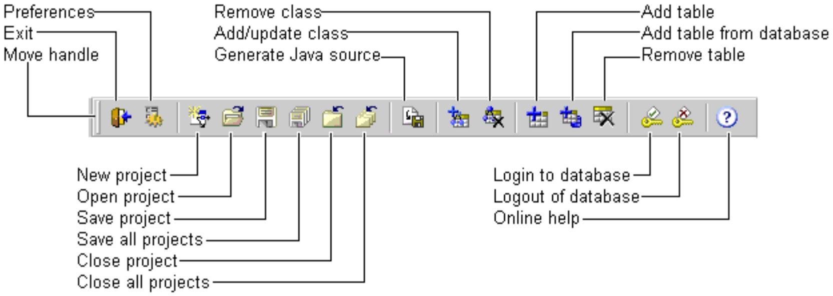
- Standard Toolbar
- Mapping Toolbar

Use these toolbars to select options and functions.

Standard Toolbar

The standard toolbar provides quick access to the standard (**File**, **Edit**, **Selected**, etc.) menu options.

Figure 1–4 Standard Toolbar

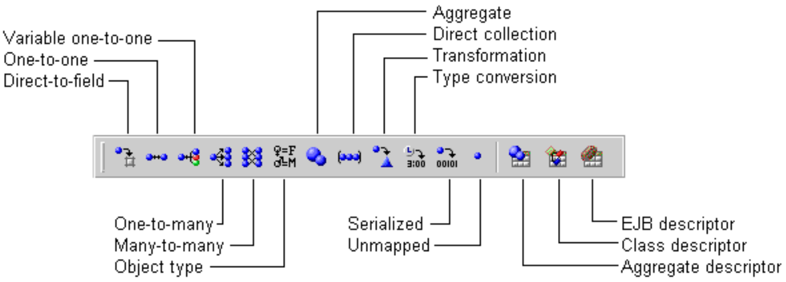


Mapping Toolbar

The mapping toolbar provides quick access to create mapping and descriptor types. You can specify a mapping or descriptor type by selecting the object from the **Project Tree** pane then clicking on the appropriate button in the mapping toolbar.

You can also right-click on the object and select the appropriate mapping from the pop-up menu.

Figure 1–5 Mapping Toolbar

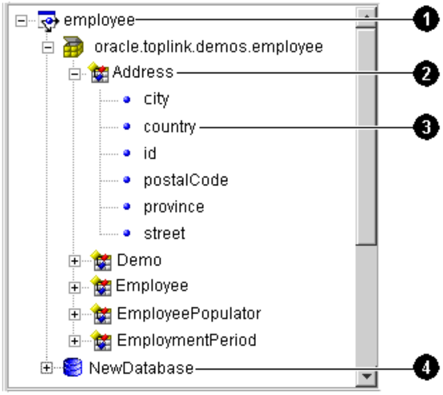


To move a toolbar, click on a blank area of the toolbar and drag it to your desktop. To re-dock the toolbar to the Mapping Workbench, click on the toolbars's move handle ‖ and drag the toolbar back to the Mapping Workbench.

Using the Project Tree Pane

TopLink displays each project's descriptors, mappings, and database tables in the **Project Tree** pane on the left side of the workbench.

Figure 1–6 Sample Project Tree Pane



User-interface components called out in Figure 1–6:

1. Project
2. Descriptor
3. Attribute/mapping
4. Database

Click on the **+/–** next to the item, or double-click the item name, to expand/collapse the item.

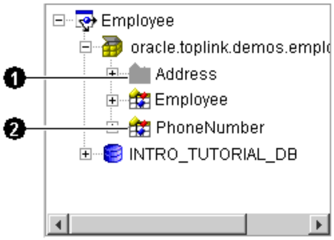
When you select an item in the **Project Tree** pane, its properties appear in the **Properties** pane (see "Using the Properties Pane" on page 1-7).

You can perform specific functions for an item by selecting the item in the **Project Tree** pane and:

- right-clicking on the object and selecting the function from the pop-up (see "Pop-up Menus" on page 1-4).
- selecting a function from the **Selected** menu (see "Menu Bar Menus" on page 1-4).

Inactive descriptors appear dimmed in the **Project Tree** pane. Inactive descriptors do not get registered with the session when the project is loaded into Java. This allows you to define and test subsets of descriptors. To activate or inactivate a descriptor, right-click on the descriptor and select **Activate/Deactivate Descriptor** from the pop-up menu.

Figure 1–7 Sample Active/inactive Descriptors



User-interface components called out in Figure 1–7:

1. Inactive descriptor
2. Active descriptor

If a descriptor contains an error (sometimes called a "neediness" message), a warning icon ⚠ appears beside the descriptor's icon in the **Project Tree** pane and a message displays in the status bar detailing the error. The *Oracle 9iAS TopLink: Troubleshooting Guide* contains complete information on each Mapping Workbench error message.

Using the Properties Pane

The **Properties** pane, on the right side of the Mapping Workbench, displays the properties associated with the currently selected item in the **Project Tree** pane.

The properties of the selected item are displayed using tab pages, grouped according to their subject.

Working with Workbench Preferences

You can customize several aspects of the TopLink Mapping Workbench.

Changing the Look and Feel

Use this procedure to customize the "look and feel" (the graphical user interface) of the Mapping Workbench.

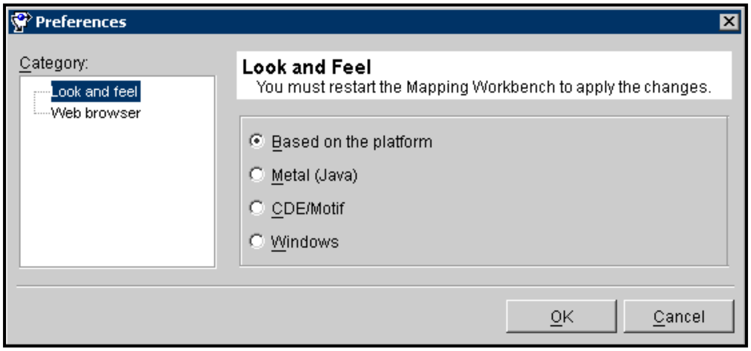
To Change the Look and Feel:

1. Click on the **Preferences** button in the toolbar. The Preferences window appears.

You can also display the preferences window by selecting **Tools > Preferences** from the menu.

2. Click on **Look and Feel** in the **Category** pane.

Figure 1–8 Look and feel Preferences



Preferences

Category:
- Look and feel
- Web browser

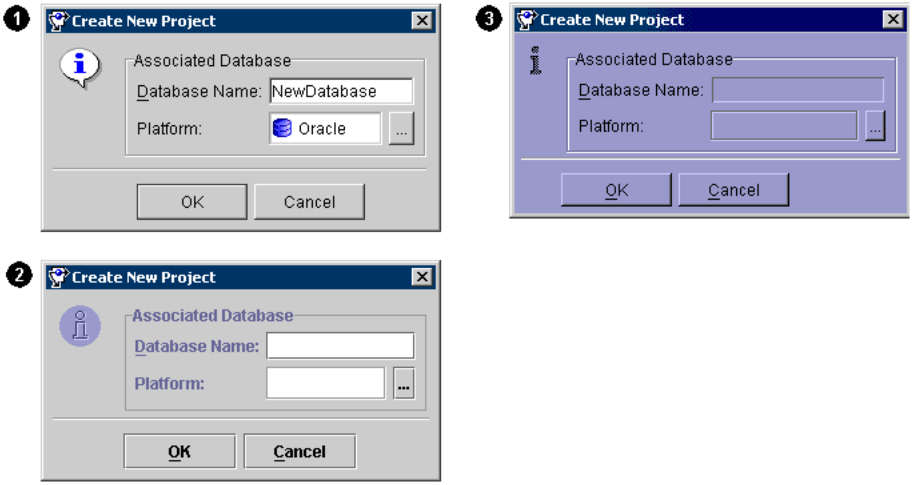
Look and Feel

You must restart the Mapping Workbench to apply the changes.

● Based on the platform
○ Metal (Java)
○ CDE/Motif
○ Windows

3. Select the look and feel and click on **OK**. You must restart the TopLink Mapping Workbench to apply the changes.

Figure 1–9 Look and Feel Samples



1 Create New Project

Associated Database
Database Name: NewDatabase
Platform: Oracle

3 Create New Project

Associated Database
Database Name:
Platform:

2 Create New Project

Associated Database
Database Name:
Platform:

User-interface "look and feel" samples in Figure 1–9:

1. Windows
2. Metal (Java)
3. CDE/Motif

Specifying a Web Browser

Use this procedure to specify a web browser to use with the Mapping Workbench. You must specify a web browser to use the online help and web-based support.

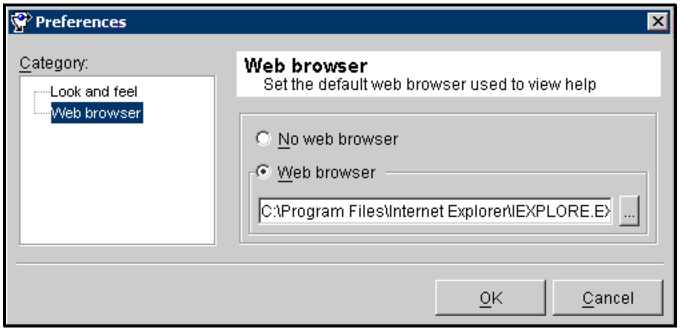
To Change the Web Browser:

1. Click on the **Preferences** button in the toolbar. The Preferences window appears.

   You can also display the preferences window by selecting **Tools > Preferences** from the menu.

2. Click on **Web Browser** in the **Category** pane.

Figure 1–10 Web Browser Preferences



Web browser

Set the default web browser used to view help

○ No web browser

● Web browser

C:\Program Files\Internet Explorer\IEXPLORE.E›

3. Select the web browser to use and click on **OK**.

Specifying Class Import Options

Use this option to specify if the Mapping Workbench verifies classes on import, when using the **Add/Refresh Class** function (see "To Update Classes from Available Packages and Classes:" on page 2-12).

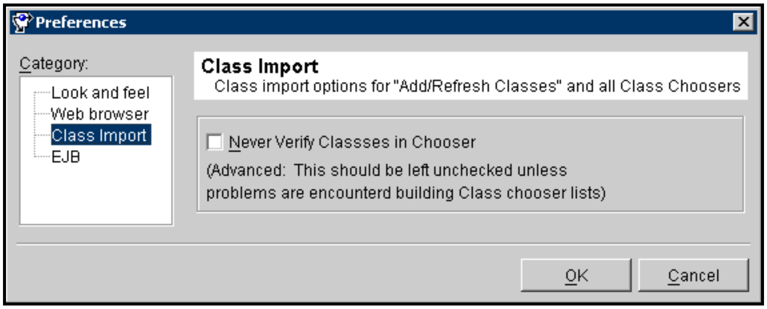
To Specify Class Import Options:

1. Click on the **Preferences** button in the toolbar. The Preferences window appears.

   You can also display the preferences window by selecting **Tools > Preferences** from the menu.

2. Click on **Class import** in the **Category** pane.

Figure 1–11 Class import Options



Class Import

Class import options for "Add/Refresh Classes" and all Class Choosers

□ Never Verify Classses in Chooser

(Advanced: This should be left unchecked unless problems are encounterd building Class chooser lists)

3. Select if the Mapping Workbench *does not* verify the classes in the chooser when performing an Add or Refresh.

> **Caution:** By default, the Mapping Workbench will always verify the classes. Select this option only if you encounter errors when displaying classes in the Select Classes window.

4. Click on **OK**.

Setting EJB Preferences

Use this procedure to specify how the Mapping Workbench updates the `ejb-jar.xml` file when saving projects.

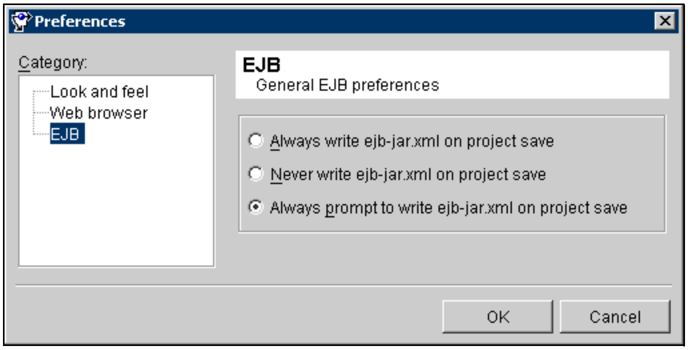
To Specify EJB Options:

1. Click on the **Preferences** button in the toolbar. The Preferences window appears.

   You can also display the preferences window by selecting **Tools > Preferences** from the menu.

2. Click on **EJB** in the **Category** pane.

Figure 1–12 EJB Options



EJB
General EJB preferences

○ Always write ejb-jar.xml on project save
○ Never write ejb-jar.xml on project save
● Always prompt to write ejb-jar.xml on project save

3. Specify if the Mapping Workbench prompts before updating the `ejb-jar.xml` file when you save the project.
4. Click on **OK**.

Working with the Mapping Workbench in a Team Environment

When using a Mapping Workbench project in a team environment, you must keep your changes “in-sync” with the other developers. See "Merging Files" on page 1-12 for more information.

You can use the Mapping Workbench with a source control system (see "Using a Source Control Management System" on page 1-11) to facilitate enterprise-level team development. If you have a small development team, you can manage the changes from within XML files (see "Sharing Project Objects" on page 1-15).

Using a Source Control Management System

If you use an enterprise, file-based, source control management system to manage your Java files, you can use the same system with TopLink Mapping Workbench. The Mapping Workbench source files are edited by the Mapping Workbench and are written out in XML file format.

The source control system’s *check-in/out* mechanism defines how to manage the source (i.e., the XML source and Mapping Workbench project file) in a multi-user environment.

To Check Out/in Mapping Workbench Project Files:

1. Check out the files from the source management system to the users system.

> **Note:** Normally, leave all locked and unlocked files in the project in **read-write** status. This allows the Mapping Workbench to updated the files, as necessary.
>
> If you know specifically which files will be changed, you can leave the remaining files in **read-only** status. The Mapping Workbench will display an error if it attempts to update a read-only file.

2. Edit the project using the Mapping Workbench.
3. Save the edited project. Some project XML files may have been altered. The source control tool will notify the user that several files have been modified locally, on their system.
4. Check-in the modified files, and add any files that have been added to the source control system for this Workbench project.

Merging Files

The most difficult aspect of team development is merging changes from two (or more) members that have simultaneously edited the same file. If one developer checks in their changes a *merge* condition exists. Usually, this condition exists in one of the root objects in the Mapping Workbench project.

Use a file comparison tool to determine the merged aspects of the project. The files to edit will vary, depending on the type of merge:

- Merging a Root File
- Merging an Aggregate File

Because a typical project may involve many changes (especially in a team environment), merging your project before checking it in may require quite a bit of development time.

Merging a Root File

These files contain references to the objects that they hold onto. The root files are:

- Project – *<projectName>*.mwp (one for each project - holds database, packages, and repository)

- Database – *<databaseName>*`.xml` (one for each project – holds tables)
- Package – *<packageName>*`.xml` (one for each package – holds descriptors)
- Class Repository – `repository.xml` (one for each project – holds classes)

Changes in these files are normally caused by adding, deleting, or renaming a **Table**, **Class**, or **Descriptor**.

To Merge a Root File:

Another developer has added a descriptor and checked-in the project while you were adding or removing descriptors from the same project.

1. Perform a file comparison on the *<packageName>*.`xml` file in merge status. The file comparison shows the addition of the descriptor XML tag and an element inside the tag.
2. Insert the XML into your *<packageName>*`.xml` file (inside the **Package** element). This brings your local code up to date to the current code in the code repository.
3. Check out any new files indicated as "missing" by your source control system. This will include the new **Descriptor** that has been added.
4. Check in all files that you have modified.

Example 1–1 Merging Projects

Another developer has added and checked in a new **Employee** class descriptor to the `com.demo` package while you were working with the package. To merge your work with the newly changed project:

- Perform a file comparison on the `com.demo.xml` file located in *<projectRoot>*`/Package/` directory that is in merge status

  The file comparison shows the addition of the descriptor XML tag and an element inside that tag:

```
<descriptor>
    <descriptor>com.demo.Employee.ClassDescriptor</descriptor>
</descriptor>
```

- Insert this XML into your `com.demo.xml` file (inside the **Package** element) to bring it up to date to the current code in the code repository.
- Check out any new files identified as "missing" by your source control system. This will include the new **Employee** class descriptor that has been added.

- Check in files that you have modified.

Merging an Aggregate File

Developers simultaneously changing the Mapping Workbench that have altered the contents of an aggregate file will also cause a merge condition. Aggregated files include:

- Class – `<className>.xml` (one for each class)
- Descriptor – `<descriptorName.type>.xml` (one for each descriptor)
- Table – `<tableName>.xml` (one for each table)

The Mapping Workbench changes these files when saving a project if you have changed any of the contents within them (such as a mapping added to a descriptor, an attribute added to a class, or a field reference changed in a table).

To Merge an Aggregate File:

If another developer has added a mapping to a descriptor and checked-in the project while you were changing a different mapping on that same descriptor:

1. Perform a file comparison on the `<descriptorName>.xml` file in merge status. The file comparison shows the addition of the Mapping XML tag and elements inside the tag.
2. Insert this XML into your `<descriptorName>.xml` file (inside the **Mappings** element). This brings your local code up to date to the current code in the code repository.
3. Check out any new files indicated as “missing” by your source control system. This will include any tables or descriptors referenced by the new mapping.
4. Check in all files that you have modified.

Example 1–2 Merging Files

Another developer has added and checked in the **firstName** mapping to the **Employee** class descriptor while you were changing a different mapping on that same descriptor.

- Perform a file comparison on the `com.demo.Employee.ClassDescriptor.xml` file located in `<projectRoot>/Descriptor/` that is in merge status

  The file comparison shows the addition of the `<Mapping>` XML tag and elements inside that tag:

```
<Mapping>
    <comment></comment>
    <descriptor>com.demo.Employee.ClassDescriptor</descriptor>
    <usesMethodAccessing>false</usesMethodAccessing>
    <inherited>false</inherited>
    <readOnly>false</readOnly>
    <instanceVariableName>firstName</instanceVariableName>
    <defaultFieldNames>
        <defaultFieldName>direct field=</defaultFieldName>
    </defaultFieldNames>
    <fieldHandle>
        <FieldHandle>
           <table>EMPLOYEE</table>
           <fieldName>F_NAME</fieldName>
        </FieldHandle>
    </fieldHandle>
    <classIndicator>BldrDirectToFieldMapping </classIndicator>
</Mapping>
```

- Insert this XML block into your `com.demo.Employee.ClassDescriptor.xml` file (inside the `<Mapping>` element) to bring it up to date to the current code in the code repository.
- Check out any new files that should be noted as missing by your source control system. This will include any tables or descriptors that may be referenced by the new mapping.
- Check in files that you have modified.

Sharing Project Objects

You can also share project objects by simply copying the table or descriptor file(s) into the appropriate directories in the target project.

After copying the file(s), insert a reference to the table or descriptor in the appropriate *<databaseName>*`.xml` or *<packageName>*`.xml` file. All references contained within these files must refer to an existing object in the project.

Managing the ejb-jar.xml File

When working in a team environment, manage the `ejb-jar.xml` file similarly to the `.xml` project files. The Mapping Workbench will edit and update the `ejb-jar.xml` file, if necessary, when working with an EJB project.

If you use a version control system, perform the same check-in/out procedures. For merge conditions, use a file comparison tool to determine which elements have been added or removed. Modify the file as necessary and check-in the file to version your work.

2

Understanding Projects

The Mapping Workbench project (`.mwp` file) stores the information about how classes map to database tables. These are language independent XML files, different from the deployment XML files generated by the Mapping Workbench, read in by the application using the `XMLProjectReader` class.

You can edit each component of project, including:

- Project settings, such as the project classpath and sequence information
- Database information, such as driver, URL, and login information
- Table schema information for the database
- Packages and classes associated with the project
- Descriptor information for each class

Working with Projects

The Mapping Workbench displays projects and their contents in the **Project Tree** pane. When you select a project, its attributes display in the **Properties** pane.

The Mapping Workbench can log runtime XML calls (in the `mw_xml.log` file) to help troubleshoot projects. Refer to the *Oracle 9iAS TopLink Troubleshooting Guide* for additional information.

Creating new Projects

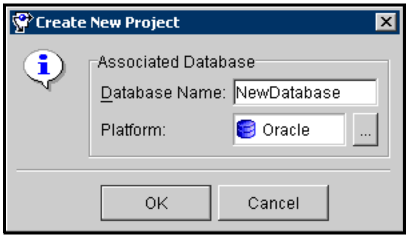
Use this procedure to create a new Mapping Workbench project.

To Create a New Project:

1. Click on the **Create New Project** button in the toolbar. The Create New Project window appears.

   You can also create a new project by selecting **File > New Project** from the menu.

Figure 2–1 Create New Project



Create New Project

Associated Database

Database Name: NewDatabase

Platform: Oracle

2. Enter the database name and platform for the new project and click on **OK**. The Save As window appears.

   See "Working with Databases" on page 3-1 for more information.

3. Select a directory location in which to save the project and click on **Save**.

> **Note:** Always use a new folder to save a project. After creating the `.mwp` project do not rename the file. See "Saving Projects" on page 2-4 to save your project with a different name.

The mapping workbench appears showing the project name in the **Project Tree** pane. Continue with "Working with Project Properties" on page 2-4 to create a project.

Opening Existing Projects

Use this procedure to open an existing project.

> **Caution:** To upgrade from a previous version of TopLink, you **must** follow specific upgrade procedures and use the TopLink Package Renamer. Refer to the TopLink Release notes and *Oracle 9iAS TopLink Getting Started Guide* for more information.

To Open an Existing Project:

1. Click on the **Open Project** button in the toolbar. The Choose a File window appears.

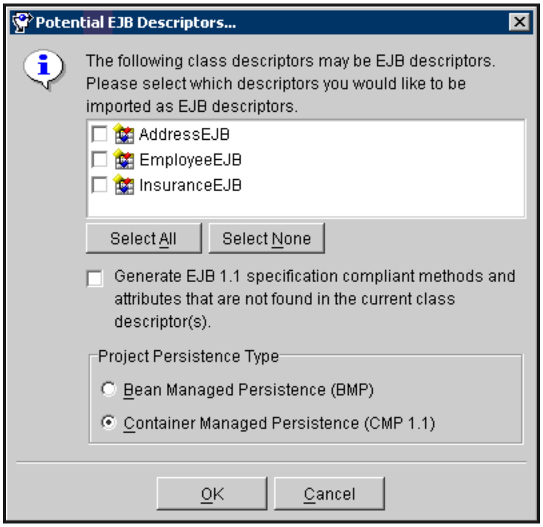
   You can also open a project by selecting **File > Open Project** from the menu.

   > **Note:** The **File** menu also contains a list of recently opened projects. You may select one of these projects to open.

2. Select the TopLink Mapping Workbench project file (`.mwp`) to open and click on **Open**. The Mapping Workbench displays the project information.

   If you open a 3.x Mapping Workbench project that contains EJBs, the Potential EJB Descriptors window appears.

Figure 2–2 Potential EJB Descriptors



The following class descriptors may be EJB descriptors. Please select which descriptors you would like to be imported as EJB descriptors.

□ AddressEJB
□ EmployeeEJB
□ InsuranceEJB

□ Generate EJB 1.1 specification compliant methods and attributes that are not found in the current class descriptor(s).

Project Persistence Type

○ Bean Managed Persistence (BMP)
● Container Managed Persistence (CMP 1.1)

3. Select which of the descriptors should be imported as EJB descriptors, the project persistence type, and click on **OK**.

   You can also specify if the Mapping Workbench generates methods and attributes that comply with the EJB specification if they are not found within the current class descriptor(s).

Saving Projects

The Mapping Workbench does not automatically save your project. Be sure to save your project often to avoid losing data.

To Save Your Project(s):

Click on the **Save Selected Project** button or **Save All Projects** button to save your project(s).

You can also save a project by selecting **File > Save Project** or **File > Save All** from the menu.

To Save Your Project with a Different Name or Location:

1. Select **File > Save As**. The Save As window appears.
2. Browse to the directory in which to save the project. In the **File Name** field, type the name of the project and click on **Save**.

> **Caution:** Do not simply rename the `.mwp` file outside of Mapping Workbench. Always rename a project by using the **Save As** option.

Refreshing the Project Tree

If the Mapping Workbench interface becomes corrupt, use the **Refresh Tree** option to redraw the **Project Tree**.

To Refresh the Project Tree:

Right-click on the project icon in the **Project Tree** pane and select **Refresh** from the pop-up panel.

You can also refresh the project tree by choosing the project icon and selecting **File > Refresh** from the menu or pressing **Ctrl+T**.

Working with Project Properties

Each project in the **Project Tree** pane contains various editable parameters. To edit the project's properties, select the project object in the **Project Tree** pane. The following tabs appear in the **Properties** pane.

- **General** project properties (persistence type and classpath)

- **Default** project properties (identity map, existence checking and field access method)
- **Sequencing**
- **Table generation** (primary key and primary key search pattern)

Working with General Project Properties

Use the project's **General** tab to specify the default persistence type and classpath information.

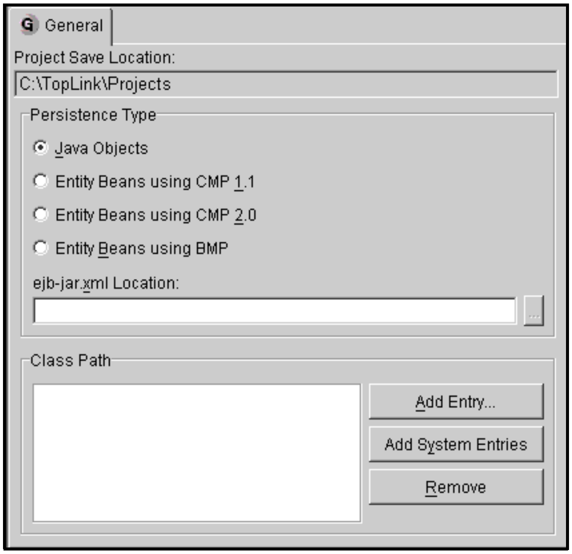
Each TopLink project uses a classpath – a set of directories, `.jar` files, and `.zip` files – when importing Java classes and defining object types.

To create a descriptor for a persistent class, the Mapping Workbench reads a compiled Java `.class` file to read its attributes and relationships.

To Specify the General Properties:

1. Choose the project object in the **Project Tree** pane.
2. Click on the **General** tab in the **Properties** pane. The **General** tab appears.

Figure 2–3 General Tab



General

Project Save Location:

C:\TopLink\Projects

Persistence Type

● Java Objects

○ Entity Beans using CMP 1.1

○ Entity Beans using CMP 2.0

○ Entity Beans using BMP

ejb-jar.xml Location:

Class Path

3. Specify the project’s persistence type. For EJB projects, you can specify the location of the `ejb-jar.xml` file. See "Mapping EJB 2.0 Entities" on page 2-6 and "Working with the ejb-jar.xml File" on page 2-15 for more information.

   Note: This field applies for EJB projects only.

4. To add a new classpath entry, click on **Add Entry** and select the directory, `.jar` file, or `.zip` file to add.

   To add the system’s classpath entries to the project, click on **Add System Entries**.

   To remove a classpath entry, select the entry and click on **Remove**.

   To create a relative classpath, select an entry and edit the path, as necessary. The path will be relative to the **Project Save Location**.

See "Working with Classes" on page 2-11 for information.

Mapping EJB 2.0 Entities

You can create a Mapping Workbench project based on information in the `ejb-jar.xml` file. Use this file to map the EJB 2.0 CMP entity beans’ virtual fields (called *Container Managed Fields*, defined by `<cmp-field>` tag) or relationships (called *Container Managed Relationship*, defined by `<cmr-field>` tag) to database tables. You must specify an `.xml` file or a `.jar` file that contains an `ejb-jar.xml` file.

The Mapping Workbench defines all descriptors for entity classes (as defined in the `ejb-jar.xml` file) as EJB descriptors. The Mapping Workbench does not create (or remove) descriptors for the interfaces and primary key class for the entity when refreshing from the `ejb-jar.xml`.

Note: The Mapping Workbench creates class descriptors for entity classes not defined in the `ejb-jar.xml` file. You must manually change the descriptor type (see "Specifying Descriptor Types" on page 4-2).

To update your project from the `.xml` file, right-click on an EJB descriptor and select **Update Descriptors from ebj-jar.xml**. You can also update the project by selecting **Selected > Update Descriptors from ebj-jar.xml** from the menu. See "Working with the ejb-jar.xml File" on page 2-15 for more information.

Working with Default Properties

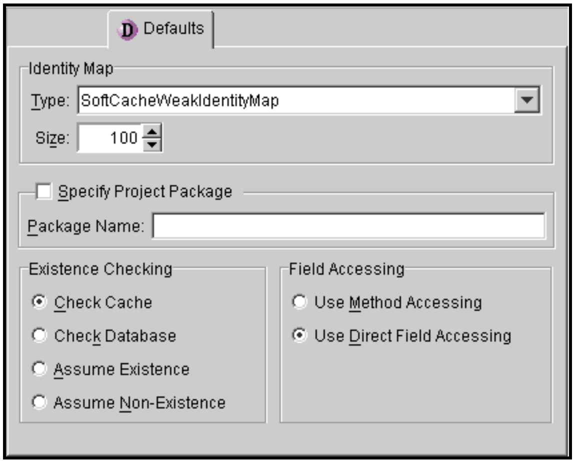
Use the project's **Default** tab to specify the default:

- identity map and existence checking policy for descriptors (if they do not have a specific identity policy)
- field accessing applied to newly created descriptors

To Specify Default Project Properties:

1. Choose the project object in the **Project Tree** pane.
2. Click on the **Defaults** tab in the **Properties** pane. The **Defaults** tab displays.

Figure 2–4 Defaults Tab



Defaults

Identity Map

Type: SoftCacheWeakIdentityMap

Size: 100

☐ Specify Project Package

Package Name:

Existence Checking

● Check Cache
○ Check Database
○ Assume Existence
○ Assume Non-Existence

Field Accessing

○ Use Method Accessing
● Use Direct Field Accessing

3. Use this table to enter data in each field:

Field	Description
Identity Map	Use the **Type** drop-down list to select the default identity map and its size for descriptors in this project (see "Working with Identity Maps" on page 4-50).
Specify Project Package	Default package to use for this project. See "Renaming Packages" on page 2-8 for information on renaming packages.
Existence Checking	Specify the type of existence checking to use.

Field	Description
Field Accessing	Specify if the descriptors use **Method** or **Direct** field accessing (see "Specifying Direct Access and Method Access" on page 4-62).

Renaming Packages

To rename your packages, you must edit each of the project's associated `.xml` files in the following sub-directories:

- **Class**
- **ClassRepository**
- **Descriptor**
- **Package**

You must also edit the package and class names in the `.mwp` file. After changing the package names in all files, open the project in the Mapping Workbench. TopLink will now use the new package name.

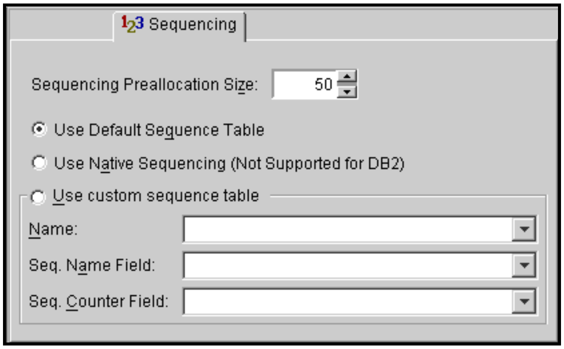
Working with Sequencing Properties

Sequence numbers are artificial keys that uniquely identify the records in a table. Use the project's **Sequencing** tab to specify default sequencing properties for all descriptors in the project.

To Specify Default Sequencing Properties:

1. Choose the project object in the **Project Tree** pane.
2. Click on the **Sequencing** tab in the **Properties** pane. The **Sequencing** tab displays.

Figure 2–5 Sequencing Tab



123 Sequencing

Sequencing Preallocation Size: 50

● Use Default Sequence Table

○ Use Native Sequencing (Not Supported for DB2)

○ Use custom sequence table

Name:

Seq. Name Field:

Seq. Counter Field:

3. Use this table to enter data in each field:

Field	Description
Sequencing Preallocation Size	Default pre-allocation size. Default is **50**.
Sequencing Type	Specify if the project uses: - **Default** sequencing - **Native** sequencing - **Custom** sequencing table
Custom Sequence Table Information	Use these fields to select the sequence table, and name and counter fields. These fields apply only when **Use Sequencing Table** is selected.

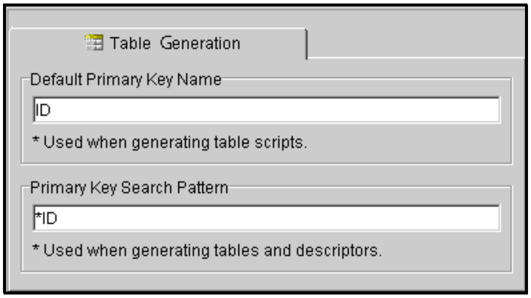
Working with Table Generation Properties

Use the project's **Table Generation** tab to specify the default primary key name and primary key search pattern (database schema) to use when generating tables. The resulting tables and columns will conform to the naming restrictions of the project's target database.

To Specify Default Table Generation Properties:

1. Choose the project object in the **Project Tree** pane.
2. Click on the **Table Generation** tab in the **Properties** pane. The **Table Generation** tab displays.

Figure 2–6 Table Generation Tab



Default Primary Key Name

ID

* Used when generating table scripts.

Primary Key Search Pattern

*ID

* Used when generating tables and descriptors.

3. Use this table to enter data in each field:

Field	Description
Default Primary Key Name	Default name to use when generating primary keys.
Primary Key Search Pattern	Default search pattern to use when generating primary keys.

Setting Default Advanced Properties

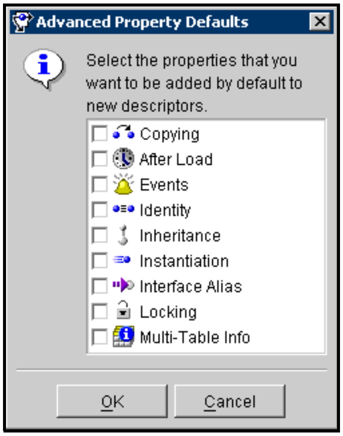
In addition to a descriptor's standard property tabs, you can specify advanced properties for each descriptor. You can also specify which of these advanced properties appear, by default, for new descriptors.

To Specify the Default Advanced Properties for Descriptors:

1. Right-click on the project object in the **Project Tree** pane and select **Set Advanced Property Defaults** from the pop-up menu. The Advanced Property Defaults window appears.

   You can also set the default advanced properties by choosing the project object and selecting **Selected > Set Advanced Property Defaults** from the menu.

Figure 2–7 Advanced Property Default



Advanced Property Defaults

Select the properties that you want to be added by default to new descriptors.

- □ Copying
- □ After Load
- □ Events
- □ Identity
- □ Inheritance
- □ Instantiation
- □ Interface Alias
- □ Locking
- □ Multi-Table Info

2. Select each advanced property to display, by default, when creating and editing descriptors.
3. Click on **OK**.

Working with Classes

The Mapping Workbench creates descriptors from Java classes and packages.

Creating Classes

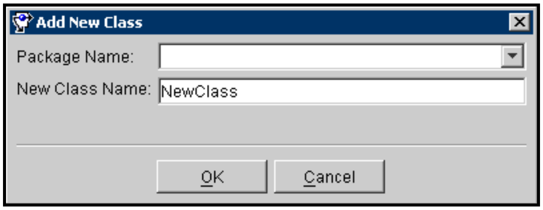
Use this procedure to create a new class and/or package from within the Mapping Workbench.

To Create a New Class:

1. Right-click on the project in the **Project Tree** pane and select **Create New Class** from the pop-up menu.

   You can also create a new class by clicking on the **Create Class** button or select the project and select **Selected > Create New Class** from the menu.

Figure 2–8 Add New Class



Package Name:

New Class Name: NewClass

2. Use the **Package Name** drop-down list to select a package or type a new package name.

3. In the **New Class Name** field enter a class name and click on **OK**. The Mapping Workbench adds the new class to your project in the **Project Tree** pane.

> **Note:** The **Class Name** must be unique within the package.

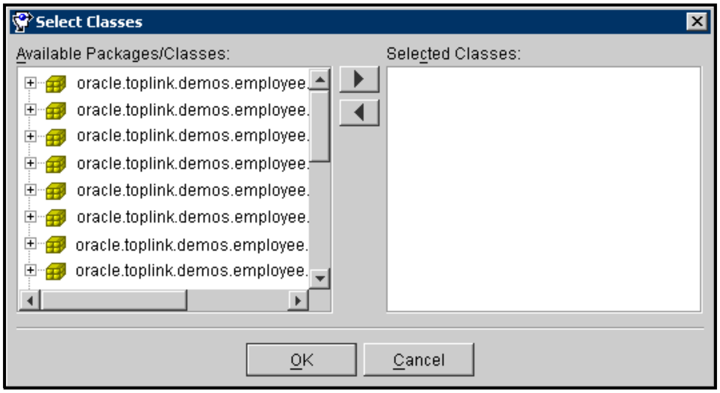
To Update Classes from Available Packages and Classes:

1. Define the available class(es) and package(s) for the project on the **General** tab. See "Working with General Project Properties" on page 2-5 for information on classes and packages.

2. Clicking on the **Add/Update Class** button. The Select Classes window appears.

   You can also update the classes by selecting **Selected > Add/Refresh Classes** from the menu.

Figure 2–9 Select Classes



oracle.toplink.demos.employee.
oracle.toplink.demos.employee.
oracle.toplink.demos.employee.
oracle.toplink.demos.employee.
oracle.toplink.demos.employee.
oracle.toplink.demos.employee.
oracle.toplink.demos.employee.
oracle.toplink.demos.employee.

3. Select the package(s) and/or class(es) to add to the project and click on **OK**. The Mapping Workbench adds the new classes to your project in the **Project Tree** pane.

> **Note:** The Mapping Workbench creates class descriptors for each package/class. You must manually change the descriptor type, if needed (see "Specifying Descriptor Types" on page 4-2).

To Remove a Class from a Project:

Select on the descriptor and click on the **Remove Class** button or select **Selected > Remove Class** from the menu.

Exporting Projects

To use your project with the TopLink Foundation Library, you must either generate deployment XML or export the project to Java source code.

Exporting Project to Java Source

Use this procedure to convert the project to Java code. Generally, this generated code executes faster and deploys easier than XML files.

To Export the Project to Java Source Code:

1. Right-click on the project in the **Project Tree** pane and select **Export Project to Java Source** from the pop-up menu. The Choose an Export File window appears.

   You can also export the project by clicking on the **Export to Java Source** button or by selecting **File > Export to Java Source** or **Selected > Export to Java Source** from the menu.

2. Select a directory location, file name (`.java`), and click on **OK**.

Exporting Table Creator Files

Use this procedure to create Java source code to generate database tables.

To Export Table Creator Files:

1. Right-click on the project in the **Project Tree** pane and select **Export Table Creator Java Source** from the pop-up menu. The Choose an Export File window appears.

   You can also export the table creator files by selecting **File > Export Table Creator Java Source** or **Selected > Export Table Creator Java Source** from the menu.

2. Select a directory location, file name (`.java`), and click on **OK**.

Generating Deployment XML

Use this procedure to generate XML files from your project that can be read by the TopLink Foundation Library. Using this option reduces development time by eliminating the need to regenerate and recompile Java code each time the project changes.

To generate deployment XML:

1. Right-click on the project in the **Project Tree** pane and select **Generate Deployment XML** from the pop-up menu. The Choose an Export File window appears.

   You can also export the project by selecting **File > Generate Deployment XML** or **Selected > Generate Deployment XML** from the menu.

2. Select a directory location, file name (`.xml`), and click on **OK**.

Working with the ejb-jar.xml File

For Mapping Workbench projects that use EJB 2.0 CMP persistence, use the `ejb-jar.xml` file to store persistence information for the application server. With the Mapping Workbench, you can import information from an existing `ejb-jar.xml` file into your project, or you can create/update the `ejb-jar.xml` from your project.

Each Mapping Workbench project uses a single `ejb-jar.xml` file. For each entities from the file you should have an EJB descriptors in the project. All entities must use the same persistence type.

As you make changes in your project, you can update the `ejb-jar.xml` file to reflect your project. Also, if you edit the `ejb-jar.xml` file outside of the Mapping Workbench, you can update your project to reflect the current file.

The following table describes how fields in the `ejb-jar.xml` file correspond to specific functions in the Mapping Workbench:

Table 2–1 ejb-jar.xml Fields and Mapping Workbench

ejb-jar.xml	Mapping Workbench
`primkey`	Bean attribute mapped to the primary key in the database table (see "Setting Descriptor Information" on page 4-5)
`ejb-name, prim-key-class, local, local-home, remote, home, and ejb-class`	EJB descriptor information on the **EJB Info** tab (see "Displaying EJB descriptor Information" on page 4-16)
`abstract-schema-name`	**Descriptor Alias** field on the **Name Queries** tab (see "Named Queries" on page 4-14)
`cmp-field`	Non-relational attributes on the **Descriptor Info** tab (see "Setting Descriptor Information" on page 4-5)
`cmp-version`	**Persistence Type** field on the **General** tab (see "Working with General Project Properties" on page 2-5) The `persistence-type` is set to `container`.
`query`	Queries listed in **Queries** tab (see "Specifying Queries" on page 12) **Note:** The **findByPrimaryKey** query is not in the `ejb-jar.xml` file, as per the EJB 2.0 specification.

Table 2–1 ejb-jar.xml Fields and Mapping Workbench

ejb-jar.xml	Mapping Workbench
`relationships`	One-to-one, one-to-many, and many-to-many mappings (see "Working with Relationship Mappings" on page 6-2)

Writing to the ejb-jar.xml File

Use this procedure to update the `ejb-jar.xml` file, based on the current Mapping Workbench information. Use the EJB preferences to specify if the Mapping Workbench automatically updates the `ejb-jar.xml` file when you save the project.

> **Note:** You can also write the information to a `.jar` file. The Mapping Workbench will automatically place the `ejb-jar.xml` file in the proper location (`META-INF/ejb-jar.xml`).

To Write the ejb-jar.xml File:

Select **Selected > Write Project to ejb-jar.xml** from the menu. You can also write the `ejb-jar.xml` file by right-clicking on the project in the **Project Tree** pane and select **Write Project to ejb-jar.xml** from the pop-up menu.

- If the project does not currently contain an `ejb-jar.xml`, the system prompts you to create a new file.
- If the system detects that changes were made to the `ejb-jar.xml` file but not yet read into the Mapping Workbench (i.e., you changed the file outside of the Mapping Workbench), the system prompts you to read the file before writing the changes.

Reading from the ejb-jar.xml File

Use this procedure to read the `ejb-jar.xml` information and update your Mapping Workbench project.

> **Tip:** To automatically create EJB descriptors in the Mapping Workbench for all entities, read the `ejb-jar.xml` file *before* adding any classes in the Mapping Workbench.

To read the ejb-jar.xml file:

Select **Selected > Update Project from ejb-jar.xml** from the menu. You can also read the `ejb-jar.xml` file by right-clicking on the project in the **Project Tree** pane and select **Update Project from ejb-jar.xml** from the pop-up menu.

3

Understanding Databases

When you create a descriptor for a class, the Mapping Workbench retrieves the table information from the database.

Working with Databases

Each Mapping Workbench project contains a database. You can create multiple logins for each database.

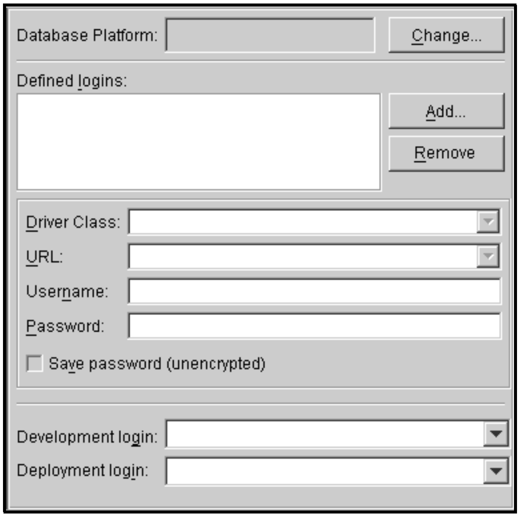
Database Properties

Use the **Database** properties to specify information about the database and login(s).

To specify the database properties:

1. Click on the database object in the **Project Tree** pane. The database properties appear in the **Properties** pane.

Figure 3–1 Database Properties



2. Use this table to enter data in each field:

Field	Description
Database Platform	Database platform for the project. Click on **Change** to select a new database.
Defined Logins	Login used to access the database. Click on **Add** to add a new login or **Remove** to delete an existing login.
Login Fields:	To edit these fields, first select a **Defined Login**.
Driver Class **URL**	The Mapping Workbench connects to databases through JDBC. Contact your database administrator for information on installing and configuring your driver.
Username	Name required to log into the database.
Password	Password required to log into the database.
Save Password	Specify if the Mapping Workbench saves the **Password** for this **Defined Login**.
Development Login **Deployment Login**	The Mapping Workbench supports multiple logins. Select a **Defined Login** to use for development and/or deployment.

3. After entering the information, continue with "Logging into the Database" on page 3-3.

Logging into the Database

You must log into the database before importing or exporting table information.

To log into the database:

Click on the database object in the **Project Tree** pane and click on the **Login** button in the toolbar. The Mapping Workbench logs into the database. The database object in the **Project Tree** pane changes to .

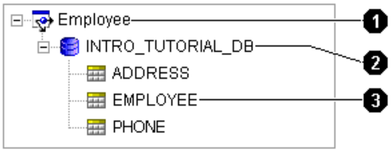
You can also log into the database by right-clicking on the database object and selecting **Log In** from the pop-up menu or **Selected > Log in** from the menu.

> **Note:** f you have not defined a login, the system displays a warning message. See "Database Properties" on page 3-1 for more information on creating a database login.

Working with Database Tables in the Project Tree Pane

When you expand the database object in the **Project Tree** pane, the Mapping Workbench displays the database tables associated with the project. You can associate tables by importing them from the database or by creating them within the Mapping Workbench.

Figure 3–2 Sample Database Tables



Database pane icons called out in Figure 3–2:

1. Project
2. Database
3. Database table

Each database table contains the following tabs in the **Properties** pane:

- **Fields** – Add or modify the table's fields, and specify the field's properties
- **References** – Specify references between tables

Creating New Tables

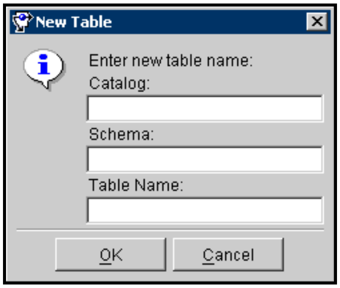
Use this procedure to create a new database table within the Mapping Workbench.

To create a new table:

1. Select the database object in the **Project Tree** pane and click on the **Add New Table** button ▦. The New Table window appears.

   You can also create a new table by right-clicking on the database object and selecting **Add New Table** from the pop-up menu or **Selected > Add New Table** from the menu.

Figure 3–3 New Table



2. Use this table to enter data in each field.

Field	Description
Catalog **Schema**	Use these fields to identify specific database information for the table. Consult your database administrator for more information.
Table Name	Name of this database table.

3. Enter the necessary information click on **OK**. The Mapping Workbench adds the database table to the project.

Note: Refer to "Generating Tables on the Database" on page 3-15 to add the table information to the database.

Continue with "Working with Database Tables in the Properties Pane" on page 3-8 to use these tables in your project.

Importing Tables from Database

The Mapping Workbench can automatically read the schema for a database and import the table data into the project.

JDBC Driver Requirements

To retrieve table information from the database, the database driver must support the following JDBC methods:

- `getTables( )`
- `getTableTypes( )`
- `getImportedKeys( )`
- `getCatalogs( )`
- `getPrimaryKeys( )`

To import tables from the database:

1. Select the database object in the **Project Tree** pane and click on the **Add/Update Existing Tables from Database** button. The Import tables from database window appears.

   You can also import tables from the database by right-clicking on the database object in the **Project Tree** and selecting **Add/Update Existing Tables from Database** from the pop-up menu or **Selected > Add/Update Existing Tables from Database** from the menu.

Figure 3–4 Import tables from Database

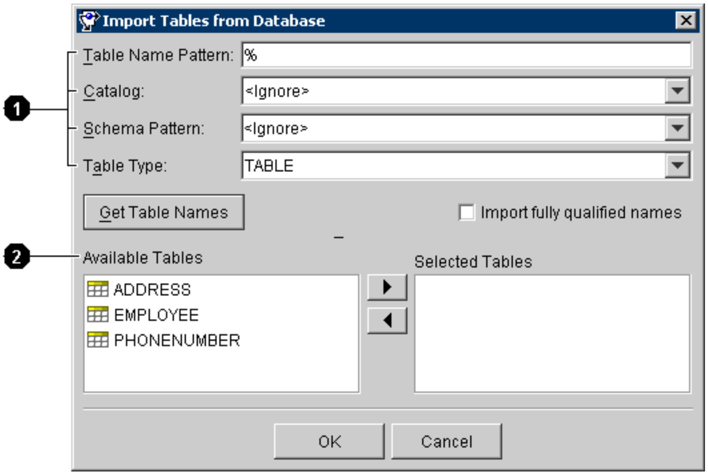


Table Name Pattern: %

Catalog: <Ignore>

Schema Pattern: <Ignore>

Table Type: TABLE

☐ Import fully qualified names

Available Tables
- ADDRESS
- EMPLOYEE
- PHONENUMBER

User-interface components called out in Figure 3–4:

1. Filters
2. Database tables that match the filters

2. Use this table to enter data in each filter field on the window:

Field	Description
Table Name Pattern	Name of database table(s) to import. Use % (percent character) as a wildcard. Tables that match the **Table Name Pattern** can be imported.
Catalog	Catalog of database table(s) to import.
Schema Pattern	Schema of database table(s) to import.
Table Type	Type of database table(s) to import.
Import Fully Qualified Names	Specify if the tables names are fully qualified against the schema and catalog.

3. Enter the filter information and click on **Get Table Names**. TopLink examines the database and displays the tables that match the filters in the **Available Tables** field.

4. Select the table(s) in the **Available Tables** area to import and click on ▶. TopLink adds the table to the **Selected Tables** field.

5. Select all the tables to import, click on **OK**. TopLink imports the tables from the database into the Mapping Workbench project.
6. Examine each table's properties to verify that the imported tables contain the correct information. See "Working with Database Tables in the Properties Pane" on page 3-8.

Removing Tables

Use this procedure to remove a database table from the project.

To remove a table:

1. Click on a database table in the **Project Tree** pane and click on the **Remove Table** button in the toolbar. The Mapping Workbench prompts for confirmation.

   You can also remove a database table from the project by right-clicking on the database object and selecting **Remove** from the pop-up menu or **Selected > Remove Table** from the menu.
2. Click on **OK**. The Mapping Workbench removes the table from the project.

Note: The table remains in the database.

Renaming Tables

Use this procedure to rename a database table in the Mapping Workbench project.

To rename the table:

1. Right-click on the table in the **Project Tree** pane and select **Rename** from the pop-up menu. The Rename window appears

   You can also rename the table by choosing the table and selecting **Selected > Rename** from the menu.
2. Enter a new name and click **OK**. The Mapping Workbench renames the table.

Note: The original table name remains in the database.

Working with Database Tables in the Properties Pane

When you select a database table in the **Project Tree** pane, its properties appear in the **Properties** pane. Each database table contains the following property tabs:

- **Fields** – Add or modify the table's fields, and specify the field's properties
- **References** – Specify references between tables

Working with Field Properties

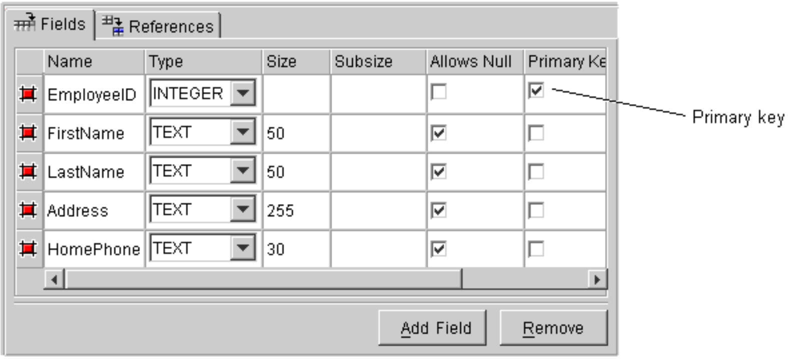
Use the database table's **Field** tab to specify properties for the database table's fields.

> **Note:** Some properties may be unavailable, depending on your database type.

To specify table field properties:

1. Select a database table in the **Project Tree** pane. The table's properties display in the **Properties** pane.
2. Click on the **Fields** tab.

Figure 3–5 Field Properties



	Name	Type	Size	Subsize	Allows Null	Primary Ke
	EmployeeID	INTEGER			□	☑
	FirstName	TEXT	50		☑	□
	LastName	TEXT	50		☑	□
	Address	TEXT	255		☑	□
	HomePhone	TEXT	30		☑	□

Primary key

3. Use this table to enter information in each field.

Field	Description
Name	Name of the field.
Type	Use the drop-down list to select the field's type. **Note:** The valid values will vary, depending on the database.
Size	Size of the field.
Subsize	Sub-size of the field.
Allows Null	Specify if this field can be null.
Primary Key	Specifies if this field is a primary key for the table.
Identity	Indicates a Sybase, SQL Server or Informix identity field.
Unique	Specifies if the value must be unique within the table.

Note: Use the scroll bar to display the additional fields.

4. Enter the necessary information for the existing fields or click on **Add Field** to add a new field.

   To remove a field, select the field and click on **Remove**.

Setting a Primary Key for Database Tables

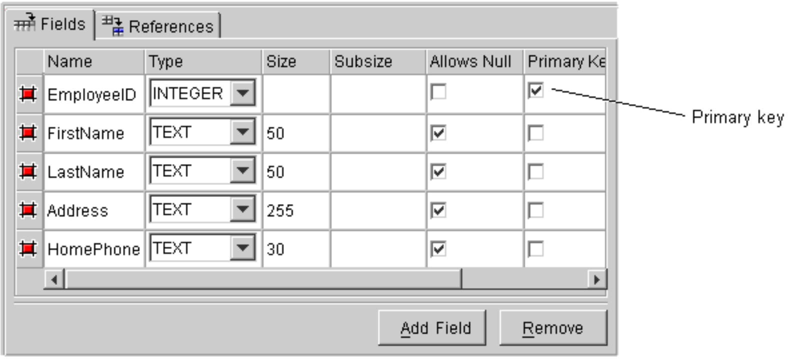
Use this procedure to set primary key(s) for a database table.

Note: The Mapping Workbench can automatically import primary key information, if supported by the JDBC driver.

To set a primary key:

1. Choose a database table in the **Project Tree** pane. Its properties appear in the **Properties** pane.
2. Click on the **Fields** tab.

Figure 3–6 Setting Primary Key for a Database Table



Name	Type	Size	Subsize	Allows Null	Primary Ke
EmployeeID	INTEGER			□	☑
FirstName	TEXT	50		☑	□
LastName	TEXT	50		☑	□
Address	TEXT	255		☑	□
HomePhone	TEXT	30		☑	□

Primary key

3. Select the **Primary Key** field(s) for the table.

Working with Reference Properties

References are table properties that contain the foreign key – they may or may not correspond to an actual constraint that exists on the database. The Mapping Workbench uses these references when you define relationship mappings and multiple table associations.

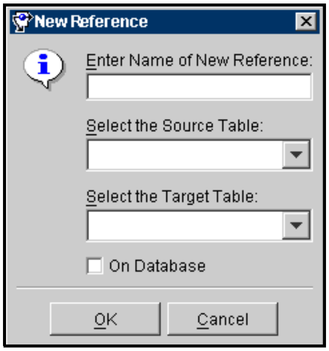
When importing tables from the database (see "Importing Tables from Database" on page 3-5), the Mapping Workbench can automatically create references (if the driver supports this) or you can define references from the workbench.

Creating table references

To create a new table reference:

1. Select a database table in the **Project Tree** pane. The table's properties displays in the **Properties** pane.
2. Click on the **Table Reference** tab.
3. In the **Table References** area, click on the **Add** button. The New Reference window appears.

Figure 3–7 New Reference Window



4. Use this table to enter information in each field.

Field	Description
Name of New Reference	Name of the reference table. If you leave this field blank, the Mapping Workbench automatically creates a name based on the format: `SOURCETABLE_TARGETTABLE`.
Select the Source Table	Name of the database table. This field is for display only.
Select the Target Table	Use the drop-down list to specify the target table for this reference.
On Database	Specify if you want to create the reference on the database when you create the table. Not all database drivers support this option.

Continue with Creating Field References.

Creating Field References

To specify table reference properties:

1. Select a database table in the **Project Tree** pane. The table's properties display in the **Properties** pane.
2. Click on the **Table Reference** tab.

Figure 3–8 References properties

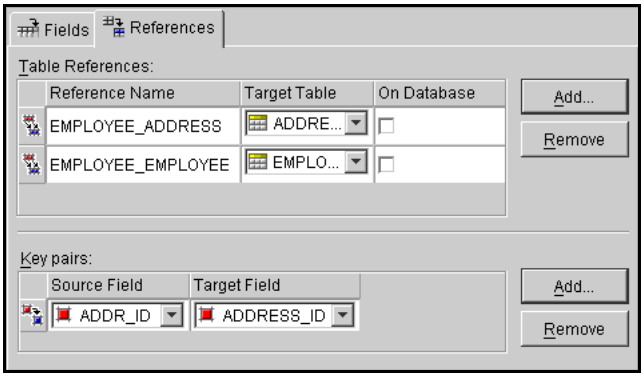


Table References:

Reference Name	Target Table	On Database
EMPLOYEE_ADDRESS	ADDRE...	□
EMPLOYEE_EMPLOYEE	EMPLO...	□

Key pairs:

Source Field	Target Field
ADDR_ID	ADDRESS_ID

3. In the **Table references** area, select a Table Reference (see "Creating table references" on page 3-10).

4. In the **Key Pairs** area, click on the **Add** button. The **Source** and **Target** fields appear on the tab.

5. Use the **Source Field** and **Target Field** drop-down lists to select the key pair for this reference.

Generating Data from Database Tables

The Mapping Workbench can automatically generate the following information from the database tables.

- SQL scripts
- Descriptors and classes
- EJB entities

You can also generate database tables from descriptors in your project.

Generating SQL Creation Scripts

Use this procedure to automatically generate SQL scripts to create the tables in a project.

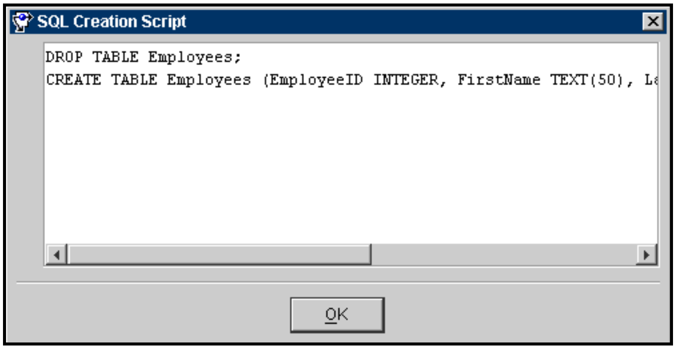
To generate SQL scripts from database tables:

1. Select the database table(s) in the **Project Tree** pane.

2. Right-click on the table(s) and select **Generate Creation Script for > Selected Table** or **All Tables** from the pop-up menu. The SQL Creation Script window appears.

   You can also generate SQL scripts by selecting **Selected > Generate Creation Script for > Selected Table** or **All Tables** from the menu.

Figure 3–9 SQL Creation Script



```
DROP TABLE Employees;
CREATE TABLE Employees (EmployeeID INTEGER, FirstName TEXT(50), La
```

3. Copy the script from the window and paste it into a file. You may need to edit the file to include additional SQL information that the Mapping Workbench could not generate.

> **Note:** If TopLink cannot determine how a particular table feature should be implemented in SQL, it generates a descriptive message in the script.

Generating Descriptors and Classes from Database Tables

The Mapping Workbench can automatically generate Java class definitions, descriptor definitions, and associated mappings from the information in database tables. You can later edit the generated information, if necessary.

For each table, the Mapping Workbench will:

- Create a class definition and a descriptor definition.
- Add attributes to the class for each column in the table.
- Automatically generate access methods, if specified.

- Create direct-to-field mappings for all direct (non-foreign key) fields in the table.
- Create relationship mappings (one-to-one and one-to-many) if there is sufficient foreign key information. You may be required to determine the exact mapping type.

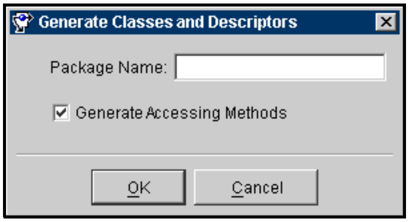
> **Note:** Class and attribute names are generated based on the table and column names. You may edit the class properties to change their names.

To generate descriptors and classes from database tables:

1. Select the database table(s) in the **Project Tree** pane.
2. Right-click on the table(s) and select **Generate Descriptors and Classes from > Selected Table** or **All Tables** from the pop-up menu. The Save Project dialog box appears.

   You can also generate SQL scripts by selecting **Selected > Generate Descriptors and Classes from > Selected Table** or **All Tables** from the menu.
3. Click on **Yes**. The Generate Classes and Descriptors dialog box appears.

Figure 3–10 Generate Classes and Descriptors



Generate Classes and Descriptors

Package Name:

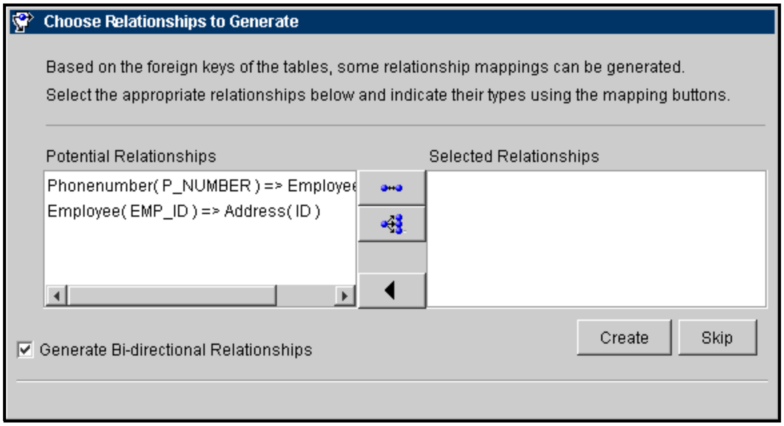
☑ Generate Accessing Methods

4. Use this table to enter data in each field:

Field	Description
Package Name	Name of package to generate
Generate Accessing Methods	Specify if the Mapping Workbench generates accessing methods for each class and descriptor

5. Enter the information and click on **OK**. If the table contains foreign key fields that may represent relationship mappings, the Choose Relationships to Generate window appears.

Figure 3–11 Choose Relationships to Generate



Choose Relationships to Generate

Based on the foreign keys of the tables, some relationship mappings can be generated.
Select the appropriate relationships below and indicate their types using the mapping buttons.

Potential Relationships

Phonenumber(P_NUMBER) => Employee
Employee(EMP_ID) => Address(ID)

Selected Relationships

☑ Generate Bi-directional Relationships

6. Select a **Potential Relationship** and click on **1:1 Mapping** or **1:M Mapping** button. See Chapter 6, "Understanding Relationship Mappings" for more information on mappings.

   You can also specify if the relationships are bidirectional.

7. Click on **Create** to automatically create the relationships (or **Skip** to generate the descriptors *without* creating these relationships.).

The newly created descriptors appear in the **Project Tree** pane of the Mapping Workbench.

Generating Tables on the Database

Use this procedure to create a table on the database, based on the information in the Mapping Workbench.

To create a table on the database:

1. Select the database table(s) in the **Project Tree** pane.
2. Right-click on the table(s) and select **Create on Database > Selected Table** or **All Tables** from the pop-up menu. The Save Project dialog box appears.

You can also create tables by selecting **Selected > Create on Database > Selected Table** or **All Tables** from the menu.

> **Note:** You must log into the database before creating tables. See "Logging into the Database" on page 3-3 for more information.

The Mapping Workbench creates the tables on the database.

Generating EJB Entities from Database Tables

Use this procedure to automatically generate EJB classes and descriptors for each database table. Generating EJB entities allows you to create:

- One EJB descriptor that implements the `<javax.ejb.EntityBean>` interface and four EJB 1.1 classes for each table
- Bean relation attributes (CMP or BMP)
- Java source fore each class
- EJB-compliant method stubs

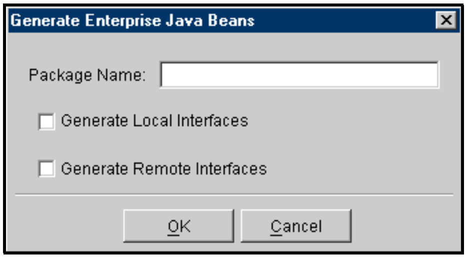
> **Note:** This option is available only for projects with CMP or BMP persistence.

To generate EJB entities:

1. Select the database table(s) in the **Project Tree** pane.
2. Right-click on the table(s) and select **Generate EJB Entities and Descriptors from > Selected Table** or **All Tables** from the pop-up menu. The Save Project dialog box appears.

   You can also create entities by selecting **Selected > Generate EJB Entities and Descriptors from > Selected Table** or **All Tables** from the menu.
3. Click on **Yes** to save your project before generating EJB entities. The Generate Enterprise Java Beans window appears.

Figure 3–12 Generate Enterprise Java Beans



Generate Enterprise Java Beans

Package Name:

□ Generate Local Interfaces

□ Generate Remote Interfaces

4. Enter a package name, select any persistence type options, and click on **OK**.

> **Note:** The **Generate Local/Remote Interfaces** options appear for 2.0 CMP and BMP projects only.

5. If the table contains foreign key fields that may represent relationship mappings, the Choose Relationships To Generate window appears. Select a potential relationship and click on the **1:1 Mapping** or **1:M Mapping** button.

   You can also specify if the relationships are bi-directional.

6. Repeat step 5 for all appropriate sets of tables.
7. Click **Create** to generate the relationship mappings (or **Skip** to generate the EJB descriptors *without* creating these relationships.).

The system creates the remote primary key, home, and bean classes for each bean and adds this information to the project. The newly created descriptor(s) appear in the **Project Tree** pane of the Mapping Workbench. Use the **EJB Info** tab to modify the EJB information.

4

Understanding Descriptors

TopLink uses *descriptors* to store the information that describes how an instance of a particular class can be represented in a relational database. Most descriptor information can be defined by the Mapping Workbench and read from a project file to be registered with a TopLink Mapping Workbench session.

For complete information on the Oracle 9*i*AS TopLink API, refer to the online API guide installed in the default TopLink directory.

> **Note:** In this document, *descriptors* refers to TopLink descriptors; *deployment descriptors* refers to EJB deployment descriptors.

Working with Descriptors

A descriptor stores all of the information describing how an instance of a particular class can be represented in a relational database. The Mapping Workbench reads a project `.mwp` file to load all of a project's information (including descriptor information).

You may need to amend a descriptor (for example, to specify a property not supported by the Mapping Workbench) after reading a project file (see "Amending Descriptors After Loading" on page 4-18). However, do not modify any descriptors after registering them with the session.

TopLink descriptors contain the following information:

- The persistent Java class it describes and the corresponding database table(s) for storing instances of that class
- A collection of mappings, which describe how the attributes and relationships for that class are stored in the database

- The primary key information of the table(s)
- A list of query keys (or aliases) for field names
- Information for sequence numbers
- A set of optional properties for tailoring the behavior of the descriptor, including support for identity maps, optimistic locking, the Event Manager, and the Query Manager
- Caching refresh options

Understanding Persistent Classes

Any class that registers a descriptor with a TopLink Mapping Workbench database session is called a *persistent class*. TopLink *does not* require that persistent classes provide public accessor methods for any private or protected attributes stored in the database.

See Appendix A, "Object Model Requirements" for more information on persistent classes object model requirements.

Specifying Descriptor Types

TopLink descriptors can be a class descriptor, an aggregate descriptor, or an EJB descriptor. After creating a descriptor, use this procedure to change the descriptor type.

> **Note:** An EJB descriptor cannot be an aggregate.

To specify a descriptor's type:

1. Choose the descriptor in the **Project Tree** pane.
2. Click on the appropriate descriptor icon (**Class**, **Aggregate**, or **EJB**) in the mapping toolbar.

   You can also specify descriptor type by choosing the descriptor and selecting **Selected > Descriptor Type >** *specific descriptor* type from the menu or by right-clicking on the descriptor in the **Project Tree** pane and selecting **Descriptor Type >** *specific descriptor type* from the pop-up menu.

Note: EJB 2.0 descriptors are created only by reading the ejb-jar.xml file. See "Mapping EJB 2.0 Entities" on page 2-6 for more information.

When changing a descriptor's type, the Mapping Workbench will add or remove property tabs, as needed.

- Converting a class or EJB descriptor to an aggregate descriptor removes the **Descriptor Info** and **Queries** tabs. Some advanced properties are not available for aggregate descriptors and will be removed from the **Properties** pane.
- Converting an aggregate descriptor to a class descriptor adds the **Descriptor Info** and **Queries** tabs.

Mapping Descriptors

Descriptors define mappings between classes and tables. To display the attributes in a specific class, expand the descriptor item in the **Project Tree** pane.

Use the mapping toolbar to choose a mapping type for each attribute.

To map a descriptor:

1. Choose a descriptor in the **Project Tree** pane. Its properties appear in the **Properties** pane.
2. On the **Descriptor Info** tab, associate the descriptor with a database table (see "Setting Descriptor Information" on page 4-5).
3. In the **Project Tree** pane, expand the descriptor to display its attributes.
4. Select an attribute and click on the appropriate mapping button in the Mapping toolbar.

Continue with "Working with Mappings" on page 4-60 to modify the mapping.

Automapping Descriptors

The Mapping Workbench can automatically map class attributes to a similarly named database field. This **Automap** function only creates mappings for unmapped attributes – it does not change previously defined mappings.

You can automap classes for an entire project or for specific tables.

> **Note:** You **must** associate a descriptor with a database table before using the **Automap** function. See "Setting Descriptor Information" on page 4-5.

To automap attributes:

To automap *all* descriptors in a project, right-click on the project icon in the **Project Tree** pane and select **Automap** from the pop-up menu or select **Selected > Automap** from the menu.

or

To automap *a specific* descriptor or attribute select the descriptor/attribute(s). Right-click and select **Automap** from the pop-up menu or select **Selected > Automap** from the menu.

Generating Java Code for Descriptors

Use this procedure to generate the Java class code for the selected descriptor or package.

To generate Java code:

1. Right-click on the descriptor or package and select **Generate Code** from the pop-up menu. The Choose a Directory dialog box appears.

   You can also generate Java code by selecting **Selected > Generate Code** from the menu.

2. Browse to the directory in which to save the Java code and click on **Save**. TopLink creates the `<DescriptorName>.java` file in the specified directory.

Working with Descriptor Properties

Each descriptor in the Mapping Workbench contains tabs and specific properties. By default, descriptors contain the following properties:

- **Descriptor Info**
- **Class Info**
- **Query Keys**
- **Queries**

- **EJB Info** (for EJB descriptors only)

Use the **Set Advanced Properties** function (see "Working with Advanced Properties" on page 4-18) to specify additional properties for each descriptor.

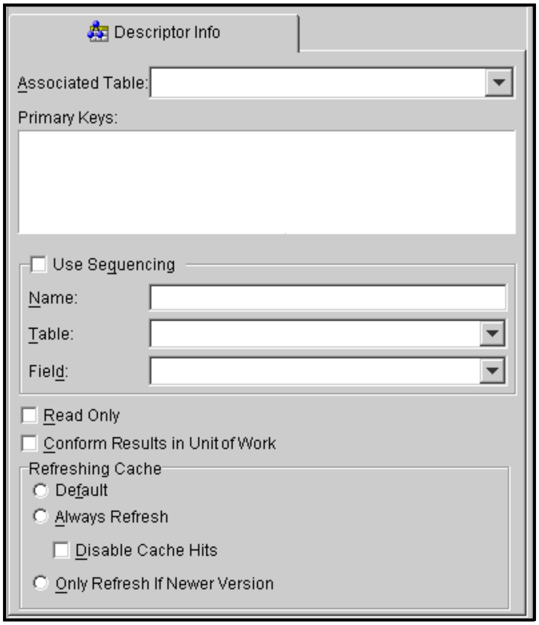
Setting Descriptor Information

Use the **Descriptor Info** tab to map a descriptor to a specific table in the database, define primary key(s), specify sequencing information, and set caching options.

To map a descriptor to a table:

1. Choose a descriptor in the **Project Tree** pane. Its properties appear in the **Properties** pane.
2. Click on the **Descriptor Info** tab.

Figure 4–1 Descriptor Info Tab



3. Use this table to enter data in each field:

Field	Description
Associated Table	Use the drop-down list to select a database table for the descriptor.
Primary Keys	Specify the primary key(s) for the table.
Use Sequencing	Specify if this descriptor uses sequencing. If selected, specify the **Name**, **Table**, and **Field** for sequencing. See "Working with Sequencing" on page 4-27 for more information.
Read Only	Specify if this descriptor is read-only.
Conform Results in Unit of Work	Specify to use the `conformResultsInUnitOfWork()` method for any read object or read all query. Refer to the *Oracle 9iAS TopLink: Foundation Library Guide* for more information.
Refreshing Cache	
Default	Use the project's default caching options.
Always Refresh	Refreshes the objects in the cache on all queries. **Note:** Using this property may impact performance.
Disable Cache Hits	Disables the cache hits on primary key read object queries.
Only Refresh if Newer	Refreshes the cache only if the object in the database is newer than the object in the cache (as determined by the **Optimistic Locking** field). See "Working with Optimistic Locking" on page 4-47 for more information.

Note: Use the caching options to specify how descriptors refresh the objects in the cache during queries. This ensures that queries against the session will refresh the objects from the row data.

Setting Class Information

After generating classes and descriptors, use the **Class** tab to:

- Rename classes, attributes, and methods
- Add, delete, or edit the generated attributes and methods
- Generate Java source to create new classes

To specify class info:

1. Choose a descriptor in the **Project Tree** pane. Its properties appear in the **Properties** pane.
2. Click on the **Class info** tab in the **Properties** pane.
3. Select the appropriate tab:
   - Class Tab
   - Attributes Tab
   - Methods Tab

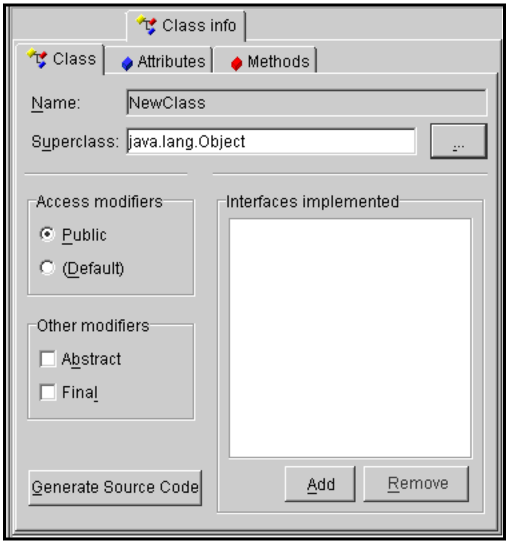
Class Tab

To add a new interface to implement, click on **Add.**

To delete an interface, select the interface and click on **Remove**.

To generate source code for the descriptor, click on **Generate Source Code**.

Figure 4–2 Class Info Tab



Name: NewClass

Superclass: java.lang.Object

Access modifiers

● Public

○ (Default)

Other modifiers

□ Abstract

□ Final

Use this table to enter data in each field:

Field	Description
Name	Name of the class. This field is for display only.
Superclass	Click on the **Browse** button and select a class and package.
Access Modifiers	Specify if the class is accessible publicly or only within its own package. Non-public classes are not accessible to the Mapping Workbench.
Other Modifiers	Specify if the class is **Final** and/or **Abstract**. Final classes are not included in the superclass selection lists for other classes to extend.
Interfaces Implemented	To add an interface, click on **Add** and select the interface and package.

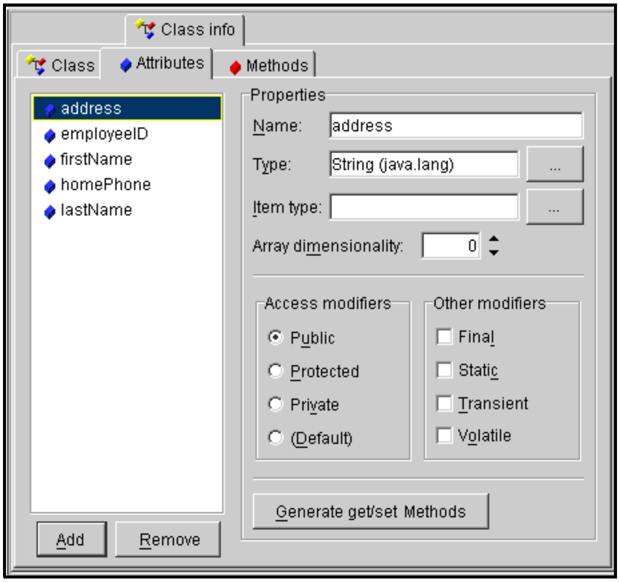
Attributes Tab

To add a new attribute, click on **Add.**

To delete an existing attribute, select the attribute and click on **Remove**.

To generate a `get` or `set` method for an attribute, click on **Generate Get/Set Methods**.

Figure 4–3 Attributes Tab



address
employeeID
firstName
homePhone
lastName

Name: address

Type: String (java.lang)

Item type:

Array dimensionality: 0

● Public
○ Protected
○ Private
○ (Default)

□ Final
□ Static
□ Transient
□ Volatile

Select an attribute then use this table to enter data in each field.

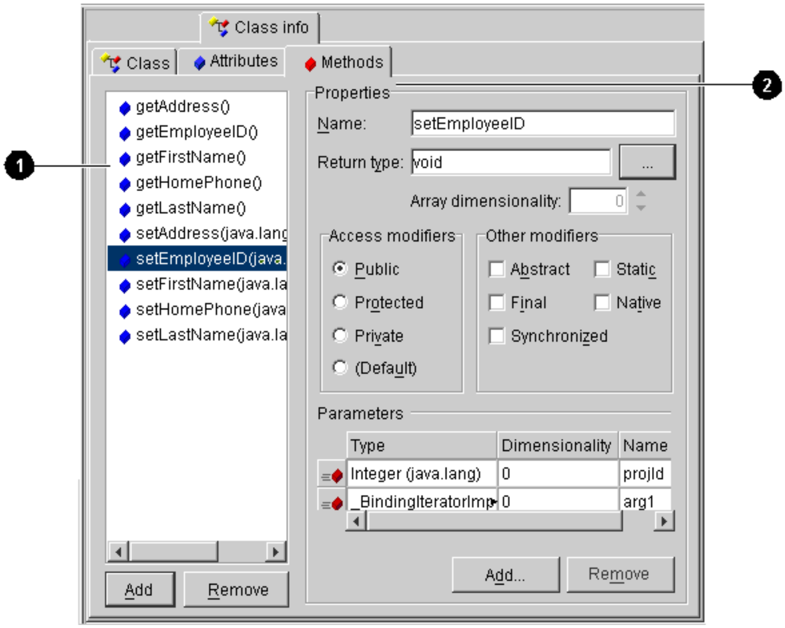
Field	Description
Name	Name of the attribute.
Type	Use the browse button to select a class and package for the attribute.
Item Type	Specify the item's type on the collection. This field applies only if **Type** is an instance of `java.util`.
Array Dimensionality	Specify the length of an array. This field applies only if **Type** is an array.
Access Modifiers	Specify how the attribute is accessible: ▪ **Public** ▪ **Protected** – Public only within its own package ▪ **Private** – Public only for subclasses ▪ **Default** – Public only within its own package
Other Modifiers	Specify if the attribute is **Final**, **Static**, **Transient**, or **Volatile**. **Note:** Selecting some modifiers may disable others.

Methods Tab

To add a new method, click on **Add.**

To delete an existing method, select the method and click on **Remove**.

Figure 4–4 Methods Tab



- getAddress()
- getEmployeeID()
- getFirstName()
- getHomePhone()
- getLastName()
- setAddress(java.lang
- setEmployeeID(java.
- setFirstName(java.la
- setHomePhone(java
- setLastName(java.la

Properties

Name: setEmployeeID

Return type: void

Array dimensionality: 0

Access modifiers: ● Public ○ Protected ○ Private ○ (Default)

Other modifiers: □ Abstract □ Static □ Final □ Native □ Synchronized

Parameters

Type	Dimensionality	Name
Integer (java.lang)	0	projId
_BindingIteratorImp	0	arg1

User-interface components called out in Figure 4–4:

1. Available methods
2. Properties of selected method

Select a method then use this table to enter data in each field:

Field	Description
Name	Name of the method.
Return Type	Use the browse button and select a class and package.
Array Dimensionality	Specify the length of the array (**Return Type**).

Field	Description
Access Modifier	Specify how the attribute is accessible: - **Public** - **Protected** – Public only within its own package - **Private** – Public only for subclasses - **Default** – Public only within its own package
Other Modifier	Specify if the attribute is **Abstract**, **Final**, **Static**, **Native**, or **Synchronized**. **Note:** Selecting some modifiers may disable others.
Parameters	Click on **Add** to include parameter(s) for the method. **Note**: The parameters are loaded in the order listed.

Query Keys

The Mapping Workbench uses query keys as an alias for a field name. With an alias, TopLink expressions can use the Java names instead of DBMS-specific field names.

Use the **Query Keys** tab to create user-defined queries or to work with automatically generated query keys.

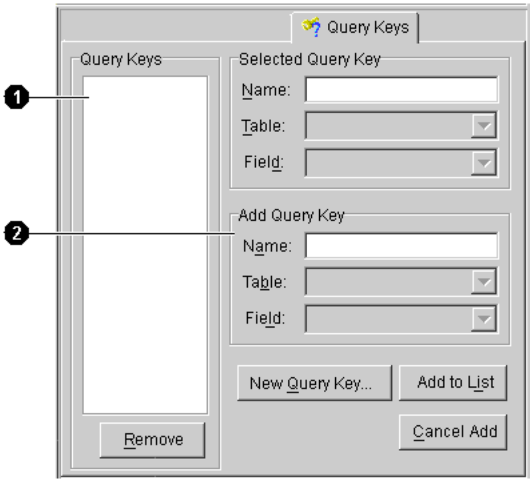
Specifying Query Keys

Use the **Query keys** tab to specify a query key for a descriptor.

To specify query keys:

1. Choose a descriptor in the **Project Tree** pane. Its properties appear in the **Properties** pane.
2. Click on the **Query Keys** tab in the **Properties** pane.

Figure 4–5 Query Keys Tab



User-interface components called out in Figure 4–5:

1. Defined query keys
2. New query key information

3. Click on **New Query Key** to create a new query.
4. Use this table to enter data in each field:

Field	Description
Name	Name of the query.
Table	Database table used by the query.
Field	Field in the table used by the query.

Specifying Queries

Use the **Queries** tab to specify EJBQL and SQL queries and finders to use for database access. The **Queries** tab contains two additional tabs: **Named Queries** and **Custom SQL**.

For 2.0 CMP projects, the `ejb-jar.xml` files stores query lists. You can define the queries in the file and then read them into the Mapping Workbench, or define them on the **Queries** tab and write them to the file. See "Writing to the ejb-jar.xml File" on page 2-16 for more information.

To create queries:

1. In the **Project Tree** pane, select a descriptor.
2. Click on the **Queries** tab in the **Properties** pane.
3. Select the appropriate tab:
   - SQL Queries
   - Named Queries

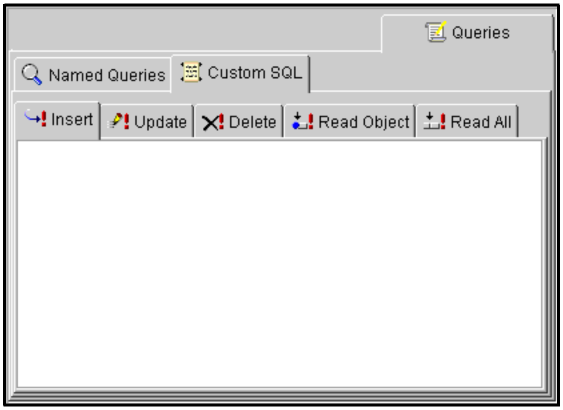
SQL Queries

Use this procedure to create custom SQL queries in the Mapping Workbench. For 2.0 CMP projects, the SQL *is not* written to the `ejb-jar.xml` file.

To create custom SQL queries:

1. In the **Project Tree** pane, select a descriptor.
2. Click on the **Queries** tab in the **Properties** pane.
3. Click on the **Custom SQL** tab.

Figure 4–6 Queries Custom SQL Tab



4. Click on the appropriate SQL function tab (**Insert**, **Update**, etc.) and type the SQL code to execute.

> **Note:** The Mapping Workbench does not validate the SQL code that you enter.

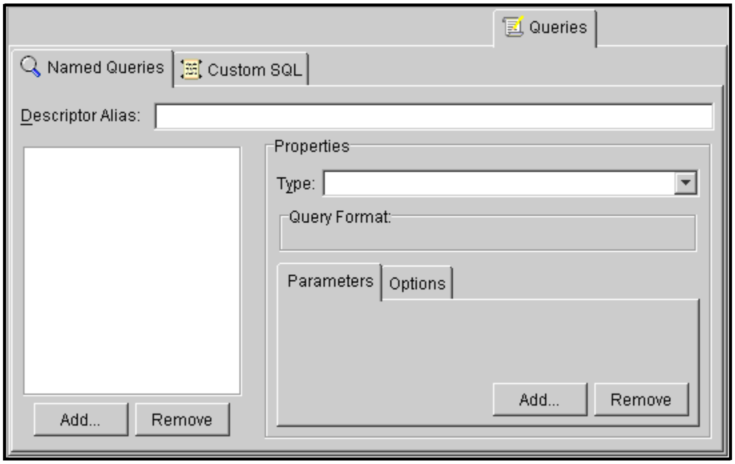
Named Queries

Use named queries to specify SQL or EJBQL queries to access the database. EJBQL is a declarative language used to present queries from an object-model perspective. Refer to the EJB specification for detailed information.

To create a named query:

1. In the **Project Tree** pane, select a descriptor.
2. Click on the **Queries** tab in the **Properties** pane.
3. Click on the **Named Queries** tab in the **Queries** tab.

Figure 4–7 Named Queries Tab



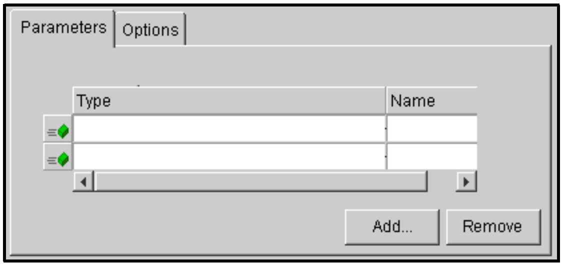
4. Click on **Add** to create a new named query. The Add Queries window appears.
5. Select the query type, enter the query name, and press **Enter**. The Mapping Workbench adds the query to the Named Query tab.
6. Use this table to enter data in each field on the tab.

Field	Description
Descriptor Alias	Alias for the descriptor class. This field applies for EJB finders only.
Name	Name of the query. The prefix of the query name specifies the query type: ■ `find` – EJB 2.0 ■ `ejbSelect` – EJB Select TopLink Reserved Finder names cannot be changed.
Type	Use the drop-down list to specify if this is a **ReadObject** or **ReadAll query.**
Query Format	Specify if this is an **EJB QL** or **SQL** query. For TopLink Reserved Finders, the query will be generated at runtime.
Query String	Enter the query string. **Note**: The mapping workbench does not validate the query string.

7. To add additional parameters to the finder, click on the **Parameters** tab and click **Add**.

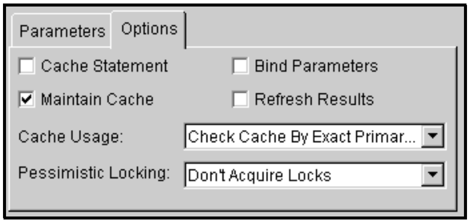
> **Note:** You can only add parameters for non-EJB descriptors.

Figure 4–8 Named Queries Parameters Tab



8. Select a class and package to add to this finder.
9. Click on the **Options** tab to add additional options.

Figure 4–9 Named Queries Options Tab



□ Cache Statement
□ Bind Parameters
☑ Maintain Cache
□ Refresh Results
Cache Usage: Check Cache By Exact Primar...
Pessimistic Locking: Don't Acquire Locks

Note: If the options on this panel are disabled, the Mapping Workbench uses the options specified in the parent.

---

10. Use this table to enter data in each field:

Field	Description
Cache Statement	Specify the `cacheStatement()` method for the query.
Bind Parameters	Specify the `bindAllParameters()` method for the query.
Maintain Cache	Specify `maintainCache()` for the query.
Refresh Results	Specify the `refreshIdentityMapResults()` method to refresh the attributes of the object resulting from the query.
Cache Usage	Select how the query checks the cache before accessing the database.
Pessimistic Locking	Select how the pessimistic locking policy acquires locks.

Note: These options are not available for `findOneByQuery` and `findManyByQuery`.

Refer to the *Oracle 9iAS TopLink Foundation Library Guide* for additional information.

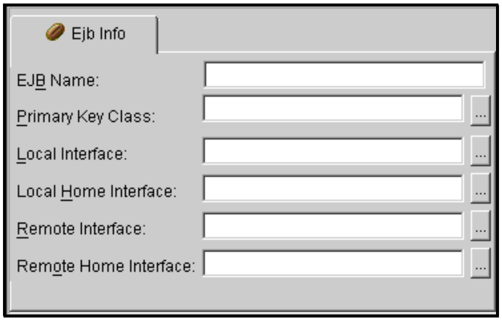
Displaying EJB descriptor Information

Use the **EJB Info** tab to display the EJB descriptor's information (from the `ejb-jar.xml` file). This tab is available only for EJB descriptors.

To display EJB descriptor information:

1. In the **Project Tree** pane, select an EJB descriptor.
2. Click on the **EJB Info** tab in the **Properties** pane.

Figure 4–10 EJB Info Tab



3. Use this table to identify each field:

Field	Description
EJB Name	Base name. When using EJB 2.0, this is specified in the **`<ejb-name>`** element of the `ejb-jar.xml` file.
Primary Key Class	Primary key. When using EJB 2.0, this is specified in the **`<prim-key-class>`** element of the `ejb-jar.xml` file.
Local Interface	Local interface. When using EJB 2.0, this is specified in the **`<local>`** element of the `ejb-jar.xml` file.
Local Home Interface	Local home interface. When using EJB 2.0, this is specified in the **`<local-home>`** element of the `ejb-jar.xml` file.
Remote Interface	Remote interface. When using EJB 2.0, this is specified in the **`<remote>`** element of the `ejb-jar.xml` file.
Remote Home Interface	Remote interface. When using EJB 2.0, this is specified in the **`<home>`** element of the `ejb-jar.xml` file.

Note: When using EJB 2.0 persistence, these fields are for display only.

Working with Advanced Properties

You can also specify the following advanced properties for each descriptor:

- Amending Descriptors After Loading
- Specifying Events
- Specifying Inheritance
- Specifying Optimistic Locking
- Specifying Multi-table Info
- Setting the Copy Policy
- Specifying Identity Mapping
- Setting Instantiation Policy
- Specifying an Interface Alias

To display advanced properties:

Right-click on a descriptor in the **Project Tree** pane and select **Set Advanced Properties >** *specific property* from the pop-up menu or select **Selected > Set Advanced Properties >** *specific property* from the menu.

Amending Descriptors After Loading

Some TopLink features cannot be configured from the workbench. To use these features, you must write a Java method to amend the descriptor after it is loaded as part of the project. This method takes the descriptor as a single parameter. You can then send messages to the descriptor or any of its specific mappings to configure advanced features.

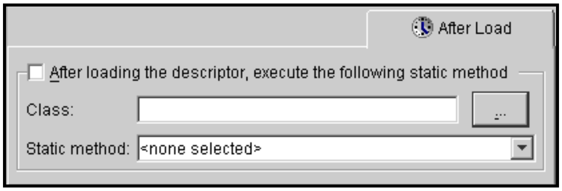
To a specify a method to execute after loading the descriptor:

1. Choose a descriptor in the **Project Tree** pane. Its properties appear in the **Properties** pane.

   If the **After load** advanced property is not visible for the descriptor, right-click on the descriptor and choose **Set Advanced Properties > After Load** from pop-up menu or from the **Selected** menu.

2. Click on the **After load** tab in the **Properties** pane.

Figure 4–11 After Load Tab



After Load

□ After loading the descriptor, execute the following static method

Class: ...

Static method: <none selected>

3. Use this table to enter data in each field:

Field	Description
After Loading...	Specify if the Mapping Workbench should execute a method after loading the descriptor.
Class	Click on the browse button and select the class of the method to execute.
Static Method	Use the **Static Method** drop-down list to select the method to execute.

Specifying Events

Use the **Events** tab to specify a descriptor's method to execute when certain events occur.

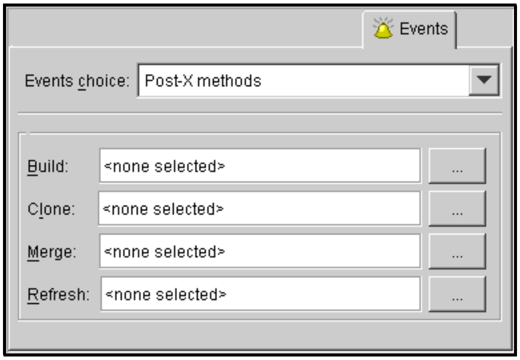
To specify an event method:

1. Choose a descriptor in the **Project Tree** pane. Its properties appear in the **Properties** pane.

   If the **Events** advanced property is not visible for the descriptor, right-click on the descriptor and choose **Set Advanced Properties > Events** from pop-up menu or from the **Selected** menu.

2. Click on the **Event** tab in the **Properties** pane.

Figure 4–12 Events Tab



Events choice: Post-X methods

Build: <none selected>

Clone: <none selected>

Merge: <none selected>

Refresh: <none selected>

3. Use this table to enter data in each field:

Field	Description
Events Choice	Select an event for this descriptor.
Methods	Select a method for each event. **Note**: The methods available will vary, depending on the selected event.

See "Supported Events" on page 4-57 for a complete list of events and methods.

Specifying Identity Mapping

TopLink Mapping Workbench specifies the default identity mapping for each descriptor in the project options (see "Working with Default Properties" on page 2-7). Use the **Identity** tab to specify identity map and existence checking information for a descriptor.

> **Note:** Changing the project's default identity policy does not affect descriptors that already exist in the project.

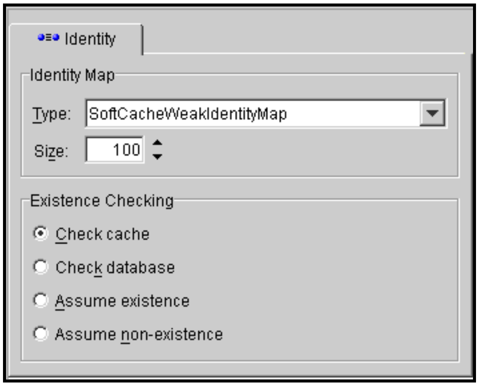
To specify an identity map for a descriptor:

1. In the **Project Tree** pane, select a descriptor.

If the **Identity** advanced property is not visible for the descriptor, right-click on the descriptor and choose **Set Advanced Properties > Identity** from the pop-up menu or from the **Selected** menu.

2. Click on the **Identity** tab.

Figure 4–13 Identity Tab



Identity

Identity Map

Type: SoftCacheWeakIdentityMap

Size: 100

Existence Checking

● Check cache

○ Check database

○ Assume existence

○ Assume non-existence

3. Use this table to enter data in each field:

Field	Description
Type	Use the **Type** drop-down list to select the identity map (see Table 4–2 for details).
Size	Size of the identity map.
Existence Checking	Specify the method of existence checking.

Specifying Inheritance

Use the **Inheritance** tab to specify the descriptor's inheritance properties as either a root or subclass (branch class or leaf class).

> **Note:** When using an aggregate descriptor in an inheritance, *all* the descriptors in the inheritance tree must be aggregates. Aggregate and Class descriptors cannot exist in the same inheritance tree.

Creating a Root Class

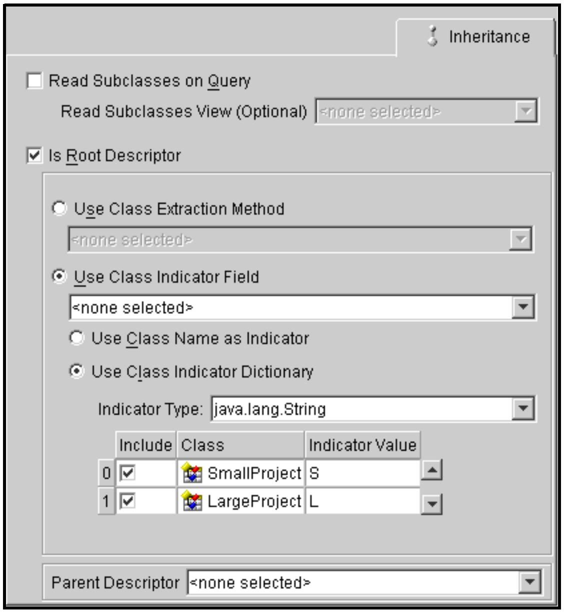
Use this procedure to create a root class.

To specify a root class:

1. In the **Project Tree** pane, choose the descriptor you wish to specify as the root.
2. Select the **Inheritance** tab in the **Property** pane.

   If the **Inheritance** tab is not visible, right-click on the descriptor and choose **Set Advanced Properties > Inheritance**.

Figure 4–14 Creating a Root Class



Inheritance

☐ Read Subclasses on Query

Read Subclasses View (Optional) <none selected>

☑ Is Root Descriptor

○ Use Class Extraction Method

<none selected>

● Use Class Indicator Field

<none selected>

○ Use Class Name as Indicator

● Use Class Indicator Dictionary

Indicator Type: java.lang.String

	Include	Class	Indicator Value
0	☑	SmallProject	S
1	☑	LargeProject	L

Parent Descriptor <none selected>

3. To instantiate the descriptor's subclasses when queried, enable the **Read Subclasses on Query** checkbox. Select a database view to use for reading subclasses if desired.

> **Note:** The view can be used for root or branch classes that have subclasses spanning multiple tables. The view must outer-join or union all of the subclass tables.

4. Enable the **Is Root Descriptor** checkbox.
5. You may use a class extraction method or a class indicator field to specify which class to instantiate on querying. Choose the option and select the appropriate method or field.
6. If you use a class indicator field, you may use the class name as the indicator, or you may use a class indicator dictionary. Choose which option you wish to use and specify the necessary information.
7. If you choose to use an indicator dictionary, choose the indicator type and set the indicator values for each subclass.

> **Note:** A list of subclasses and their indicator values appears when the subclasses have set their parent descriptor. Abstract roots are not in the list.

8. If you want instances of the subclasses to be instantiated when the root class is queried, enable the **Read Subclasses on Query** checkbox. Do not enable this checkbox for leaf classes.

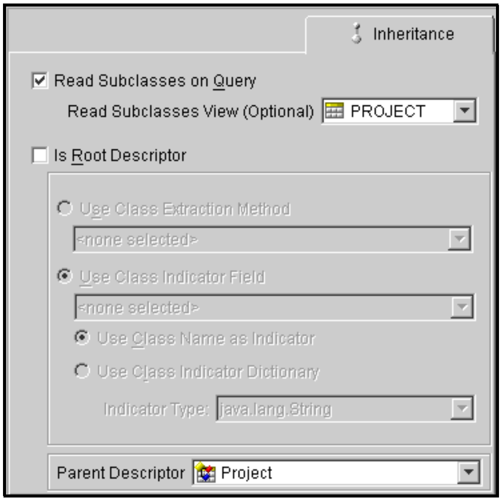
Creating Branch and Leaf Classes

After setting up the root class for inheritance, you must also specify properties for branch and leaf classes.

To create branch and leaf classes:

1. In the **Project Tree** pane, choose the descriptor for which to specify inheritance information.
2. If the **Inheritance** advanced property has not been added to the descriptor, right-click on the descriptor and choose **Set Advanced Properties > Inheritance**.
3. Select the **Inheritance** tab of the properties window.
4. Ensure that **Is Root Descriptor** is not selected. The **Parent Descriptor** drop-down list is now enabled and the class indicator information is disabled.

Figure 4–15 Creating Branch and Leaf Classes



Inheritance

☑ Read Subclasses on Query

Read Subclasses View (Optional) PROJECT

□ Is Root Descriptor

○ Use Class Extraction Method

<none selected>

● Use Class Indicator Field

<none selected>

● Use Class Name as Indicator

○ Use Class Indicator Dictionary

Indicator Type: java.lang.String

Parent Descriptor Project

5. Select the parent descriptor from the **Parent Descriptor** drop-down list. This may be the root class or a branch class.

6. Enable the **Read Subclasses on Query** option if this is a branch class and you want its subclasses to be instantiated when it is queried. Choose a database view for reading subclasses, if desired. Do not enable this checkbox for leaf classes.

Specifying Optimistic Locking

Use the **Locking** tab to specify if the descriptor uses optimistic locking.

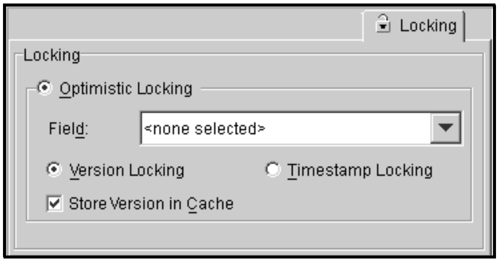
To specify a descriptor's locking policy:

1. In the **Project Tree** pane, select a descriptor.

   If the **Locking** advanced property is not visible for the descriptor, right-click on the descriptor and choose **Set Advanced Properties > Locking** from the pop-up menu or from the **Selected** menu.

2. Click on the **Locking** tab.

Figure 4–16 Locking Tab



Locking

Locking

● Optimistic Locking

Field: <none selected>

● Version Locking ○ Timestamp Locking

☑ Store Version in Cache

3. Use this table to enter data in each field:

Field	Description
Optimistic Locking	Specify if the descriptor uses optimistic locking.
Field	Use the **Field** drop-down list to select the correct field.
Version Locking	Specify the descriptor uses version locking.
Timestamp Locking	Specify the descriptor uses timestamp locking.
Store Version in Cache	Specify if you want to store the version information in the cache.

Specifying an Interface Alias

Use the **Interface Alias** tab to specify a descriptor's alias. Each descriptor may have one interface alias. Use the interface in queries and relationship mappings.

> **Note:** If you use an interface *alias*, do not associate an interface *descriptor* with the interface.

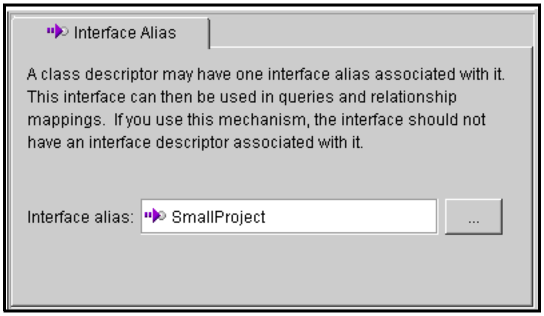
To specify an interface alias:

1. In the **Project Tree** pane, select a descriptor.

   If the **Interface Alias** advanced property is not visible for the descriptor, right-click on the descriptor and choose **Set Advanced Properties > Interface Alias** from pop-up menu or from the Selected menu.

2. Click on the **Interface Alias** tab.

Figure 4–17 Interface Alias Tab



A class descriptor may have one interface alias associated with it. This interface can then be used in queries and relationship mappings. If you use this mechanism, the interface should not have an interface descriptor associated with it.

Interface alias: SmallProject

3. Click on the browse button and select an interface.

Working with Primary Keys

A primary key is a column (or a combination of columns) that contains a unique identifier for every record in the table. In the Mapping Workbench, every table that stores persistent objects must have a primary key. Tables that require multiple columns to create an identifier use a *composite* primary key. Setting the primary key for a table also sets the primary key for the descriptor that uses the table.

The Mapping Workbench implements primary keys using sequence numbers (see "Working with Sequencing" on page 4-27).

Each descriptor must provide mappings for its primary key. These mappings may be direct, transformation or one-to-one. The Mapping Workbench does not require you to define a primary key constraint in the database – only that the fields specified for the primary key are unique.

Note: Primary keys for classes in an inheritance hierarchy or for descriptors that map to multiple tables have special requirements. Refer to"Working with Inheritance" on page 4-30 and "Working with Multiple Tables" on page 4-40 for more information.

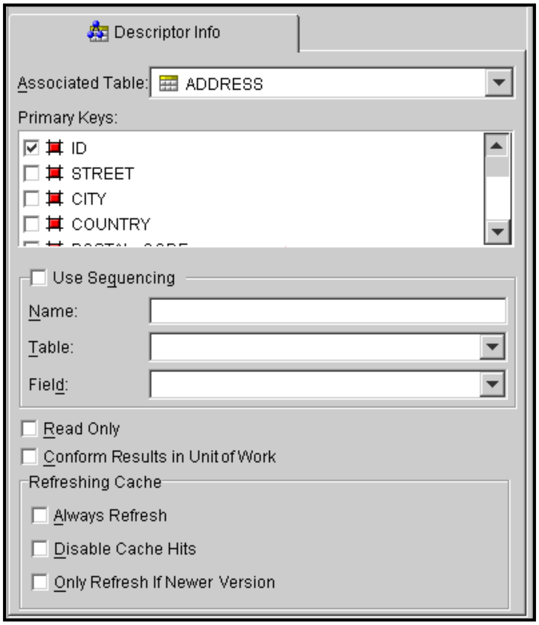
Setting a Primary Key for Descriptors

Use this procedure to set a primary key for a descriptor.

To set a primary key:

1. Choose a descriptor in the **Project Tree** pane. Its properties appear in the **Properties** pane.
2. Click on the **Descriptor Info** tab.

Figure 4–18 Primary Keys



Descriptor Info

Associated Table: ADDRESS

Primary Keys:

☑ ID
□ STREET
□ CITY
□ COUNTRY

□ Use Sequencing

Name:

Table:

Field:

□ Read Only

□ Conform Results in Unit of Work

Refreshing Cache

□ Always Refresh

□ Disable Cache Hits

□ Only Refresh If Newer Version

3. Select the field(s) to set as the primary key.

Working with Sequencing

Sequence numbers are artificial keys that uniquely identify the records in a table. When you define a sequence number field for a descriptor, the Mapping Workbench automatically generates a new sequence number every time you insert a new record into the table.

Use the project's **Sequencing** tab or the **Sequencing** area of a descriptor's **Descriptor Info** tab to specify sequencing information

Database tables often use a sequence number as the primary key. The Mapping Workbench can use the database's native support or a sequence table to maintain sequence numbers.

Tip: Oracle recommends using sequence numbers for primary keys since they are single, guaranteed, unique values.

Other data values may require composite primary keys to make up a unique value, which is less optimal. Additionally, non-artificial values may need to change, and this is not allowed for primary keys.

Using Sequence Numbers with Entity Beans

When implementing sequencing for Entity Beans, you must provide `create()` methods and the corresponding `ejbCreate()` and `ejbPostCreate()` methods for your bean home and bean class.

TopLink creates the primary key value when you first insert the bean in the database. The key value is not passed as a parameter to the `create()` methods because they do not set the primary key value (the key is generated).

Note: Be careful when using transactions with these create methods. If you create an Entity Bean within a transaction and you use native sequencing in Sybase, SQL Server or Informix, the bean's key is not initialized until the transaction commits and bean is persisted to the database for the first time.

Using Native Sequencing

Oracle, Sybase, SQL Server and Informix databases support native sequencing in which the DBMS generates the sequence numbers. However, the Mapping Workbench must still tell the DBMS to assign sequence number values.

- For Oracle databases create a SEQUENCE object in the database.
- For Sybase and SQL Server databases set the primary key field to IDENTITY.
- For Informix databases set the primary key field to use SERIAL.

Tip: If you use native sequencing in these databases, the Mapping Workbench cannot support pre-allocation. Oracle recommends using the sequence table instead. Oracle databases support pre-allocation, but only if the sequence increment matches the pre-allocation size. See "Pre-allocating Sequence Numbers" on page 4-29 for more information.

Using Sequence Tables

If your database does not use native sequencing, you must manually create the sequence table (named SEQUENCE). Use this table to store each table, as illustrated below:

Field name	Field format	Description
SEQ_NAME	CHAR	Name of the sequence number
SEQ_COUNT	NUMERIC	Current value

After creating the table, you must initialize the table within the application. The value of the `SEQ_COUNT` field for each sequence should be zero (**0**) as in the following table.

SEQ_NAME	SEQ_COUNT
EMP_SEQ	0
PROJ_SEQ	0

Pre-allocating Sequence Numbers

To increase the speed of database inserts, obtain a block of sequence numbers (by setting an allocation size) instead of executing a corresponding `SELECT` statement to obtain the newly assigned sequence number each time you create an object.

TopLink uses a default pre-allocation size of **50** when using a sequence table and **1** when using native sequencing.

- When using native sequencing in Sybase, SQL Server or Informix databases, pre-allocation cannot be set – it is always **1**.
- When using native sequencing, you must set the pre-allocation size explicitly in the Mapping Workbench.

- When using native sequencing in an Oracle database, you can use pre-allocation only if an `INCREMENT` is set on the Oracle Sequence object (not the `CACHE` option). This increment *must* match the pre-allocation size specified in the Mapping Workbench. If the increment is set incorrectly, invalid and negative sequence numbers could be generated. The `CACHE` option specifies how many sequences are pre-allocated on the database server, while the `INCREMENT` specifies the number that can be pre-allocated to the database client.

> **Tip:** Oracle recommends using sequence pre-allocation because of its performance and concurrency benefits.

Creating the Sequence Table on the Database

Normally, the database administrator defines the sequence table or sequencing object. However, you can use TopLink's schema manager to define the sequence numbers using:

```
SchemaManager schemaManager = new SchemaManager(session);

schemaManager.createSequences();
```

You should only execute this command once. The `SchemaManager` creates a sequence entry for each registered descriptor.

Refer to the *TopLink: Foundation Library Guide* for more information on using the schema manager to create number information in the database.

Working with Inheritance

Inheritance describes how a child class inherits the characteristics of its parent class. TopLink provides multiple methods to preserve the inheritance relationships. You can override mappings that have been specified in a superclass or map attributes that have not been mapped at all in the superclass.

> **Note:** When using an aggregate descriptor in an inheritance, *all* the descriptors in the inheritance tree must be aggregates. Aggregate and Class descriptors cannot exist in the same inheritance tree.

Using Inheritance with EJBs

Although inheritance is a standard tool in object-oriented modeling, the current EJB specification contains only general information regarding inheritance. You should fully understand the current EJB specification before implementing inheritance.

> **Caution:** Use caution when using inheritance. The next EJB specification may dictate inheritance guidelines that are not supported by the various servers.

Mapping Inherited Attributes in One Descriptor

If you are mapping only one class in an inheritance hierarchy, you may map attributes that it inherits from any of its superclasses.

To map attributes in one descriptor:

1. In the **Project Tree** pane, choose a descriptor.
2. Right-click on the descriptor and select **Map Inherited Attributes >** *specific location* from the pop-up menu. You can also map the attributes by choosing **Selected > Map Inherited Attributes** from the menu.

   You can map inherited attributes to:

   - Superclass
   - Root minus one
   - Selected class
3. Map the now visible attributes as though they belonged to this descriptor.

You may also do this if you have a common superclass that stores little or no persistent data. For example, if you were mapping subclasses of `java.rmi.RemoteObject`, each subclass could be mapped independently.

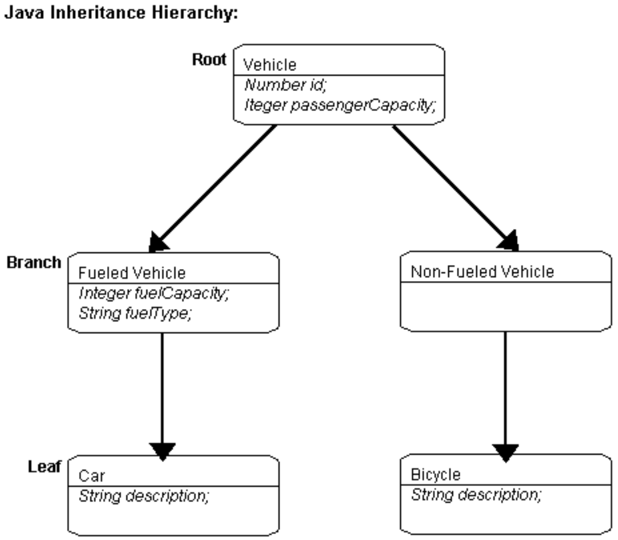
Supporting Inheritance Using One Table

You can store classes with multiple levels of inheritance in a single table to optimize database access speeds.

Example 4–1 Vehicle Object Model

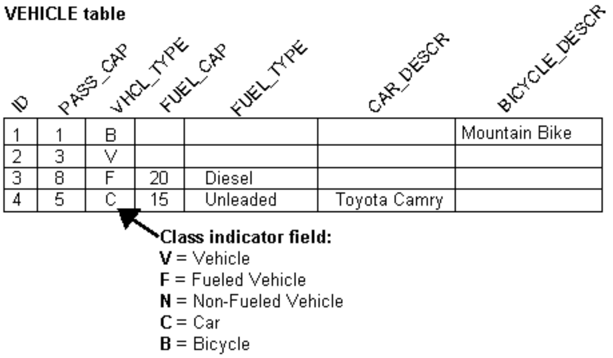
The following diagram illustrates the `Vehicle` object model.

Figure 4–19 Supporting Inheritance Using One Table



The entire inheritance hierarchy can share the same table, as in Figure 4–20. The `FueledVehicle` and `NonFueledVehicle` subclasses can share the same table even though `FueledVehicle` has some attributes that `NonFueledVehicle` does not. The `NonFueledVehicle` instances waste database resources because the database must still allocate space for the unused portion of its row. However, this approach saves on accessing time because there is no need to join to another table to get the additional `FueledVehicle` information.

Figure 4–20 Inheritance Using a Superclass Table with Optional Fields



VEHICLE table

ID	PASS_CAP	VHCL_TYPE	FUEL_CAP	FUEL_TYPE	CAR_DESCR	BICYCLE_DESCR
1	1	B				Mountain Bike
2	3	V				
3	8	F	20	Diesel		
4	5	C	15	Unleaded	Toyota Camry	

Class indicator field:
V = Vehicle
F = Fueled Vehicle
N = Non-Fueled Vehicle
C = Car
B = Bicycle

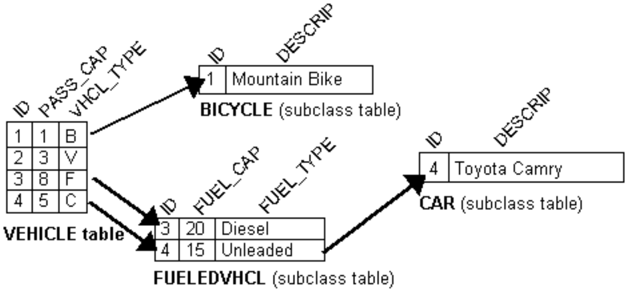
Supporting Inheritance Using Multiple Tables

For subclasses that require additional attributes, you can use multiple tables instead of a single superclass table. This optimizes storage space because there are no unused fields in the database. However, this may affect performance because TopLink must read from more than one table before it can instantiate the object. TopLink first looks at the class indicator field to determine the class of object to create, then uses the descriptor for that class to read from the subclass tables.

Example 4–2 Inheritance Example

Figure 4–21 illustrates the TopLink implementation of the `FUELEDVHCL`, `CAR`, and `BICYCLE` tables. All objects are stored in the `VEHICLE` table. `FueledVehicle`, `Car`, and `Bicycle` information is also stored in the secondary table.

Figure 4–21 Inheritance Using Separate Tables for Each Subclass



ID	PASS_CAP	VHCL_TYPE
1	1	B
2	3	V
3	8	F
4	5	C

VEHICLE table

ID	DESCRIP
1	Mountain Bike

BICYCLE (subclass table)

ID	FUEL_CAP	FUEL_TYPE
3	20	Diesel
4	15	Unleaded

FUELEDVHCL (subclass table)

ID	DESCRIP
4	Toyota Camry

CAR (subclass table)

> **Note:** Because `NonFueledVehicle` does not hold any attributes or relationships, it does not need a secondary table. For performance considerations, this design is inefficient because it requires multiple table fetching.

Finding Subclasses

An inheritance mapping for a root class must be able to locate its subclasses by using one of the following methods:

- Providing a *class indicator field*, which contains a key corresponding to its subclass
- Including a *class extraction method*, which can be implemented in Java code; this is simply a method that returns a `java.lang.Class` object
- Using class names directly in the class indicator field

Providing a Class Indicator Field

Use a *class indicator field* in the table of the root class table to indicate which subclass should be instantiated. The indicator field should not have an associated direct mapping unless it is set to read-only.

> **Note:** If the indicator field is part of the primary key, define a write-only transformation mapping for the indicator field. Refer to "Working with Transformation Mappings" on page 5-9 for more information.

You can use strings or numbers as values in the class indicator field. The root class descriptor must specify how the value in the class indicator field translates into the class to be instantiated. The following table illustrates the class indicator mapping from the `Vehicle` class containing four entries.

Table 4–1 Class Indicator Mapping from the Vehicle Class

Key	Value
F	FueledVehicle
N	NonFueledVehicle
C	Car
B	Bicycle

When working with hierarchies more than two levels deep, the class indicator field and the class indicator mapping can only be in the root class.

> **Note:** All concrete classes in the hierarchy must have a defined indicator value.

Understanding Root, Branch, and Leaf Classes in an Inheritance Hierarchy

TopLink allows three types of classes in an inheritance hierarchy:

- The root class stores information about *all* instantiable classes in its subclass hierarchy. By default, queries performed on the root class return instances of the root class and its instantiable subclasses. However, the root class can be configured so queries on it return only instances of itself without instances of its subclasses. For example, the `Vehicle` class in Example 4–1, "Vehicle Object Model" on page 4-31 is a root class.
- *Branch classes* have a persistent superclass and also have subclasses. By default, queries performed on the branch class return instances of the branch class and any of its subclasses. However, like the root class, the branch class can be configured so queries on it return only instances of itself without instances of its subclasses. For example, the `FueledVehicle` class in Example 4–1, "Vehicle Object Model" is a branch class.
- *Leaf classes* have a persistent superclass in the hierarchy but do not have subclasses. Queries performed on the leaf class can only return instances of the

leaf class. For example, the `Car` class in Example 4–1, "Vehicle Object Model" is a leaf class.

Specifying Primary Keys in an Inheritance Hierarchy

TopLink assumes that all of the classes in an inheritance hierarchy have the same primary key, as set in the root descriptor. Child descriptors associated with tables that have different primary keys must define the mapping between the root primary key and the local one.

For more information on primary keys in an inheritance hierarchy, see "Specifying Multi-table Info" on page 4-41.

Mapping Inherited Attributes in a Subclass

If you are defining the descriptor for a class that inherits attributes from another class, you can create mappings for those attributes. If you re-map an attribute that was already mapped in the superclass, the new mapping applies to the subclass only. Any other subclasses that inherit the attribute are unaffected.

To view and map attributes inherited from a superclass:

1. In the **Project Tree** pane, right-click a descriptor and choose **Map Inherited Attributes >** *selected class* from the pop-up menu or choose **Selected > Map Inherited Attributes** from the menu.

   The mappings list now includes all the attributes from the superclass of this class.

2. Map any desired attributes. See "Working with Mappings" on page 4-60.

   If you leave inherited attributes unmapped, TopLink will use the mapping (if any) from the superclass, if the superclass's descriptor has been designated as the parent descriptor.

Working with Interfaces

An *interface* is a collection of method declarations and constants used by one or more classes of objects. Domain classes can implement interfaces or can reference existing interfaces. TopLink supports interfaces in the following methods:

- In a *variable class relationship*, a domain object references another domain object or a collection of objects that implement a specific interface
- A read query can be issued to query an interface

Understanding Interface Descriptors

An *interface descriptor* is a descriptor whose reference class is an interface. Each domain class specified in TopLink has a related descriptor. A descriptor is a set of mappings that describe how an object's data is represented in a relational database. It contains mappings from the class instance variable to the table's fields, as well as the transformation routines necessary for storing and retrieving attributes. The descriptor acts as the link between the Java object and its database representation.

An interface is simply a collection of abstract behavior that other classes can use. There is no representation of interfaces on the relational database; an interface is purely a Java object concept. Therefore, a descriptor defined for the interfaces does not map any relational entities on the database.

Note: You cannot create or edit interface descriptors in the Mapping Workbench.

The components defined in the interface descriptor are:

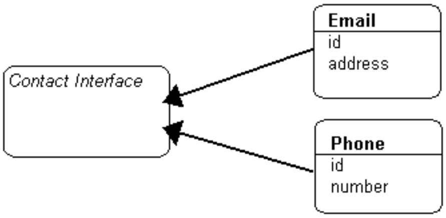
- The Java interface it describes
- The parent interface(s) it implements
- A list of abstract query keys

An interface descriptor does not define any mappings, because there is no concrete data or table associated with it. A list of abstract query keys is defined so that one can issue queries on the interfaces. A read query on the interface results in reading one or more of its implementors.

Example 4–3 Interface Examples

The following illustration shows an interface implemented by two descriptors.

Figure 4–22 Classes Implement an Interface



Following is the sample code implementation for the descriptors for `Email` and `Phone`:

```
Descriptor descriptor = new Descriptor();
    descriptor.setJavaInterface(Contact.class);
    descriptor.addAbstractQueryKey("id");
    return descriptor;
Descriptor descriptor = new Descriptor();
    descriptor.setJavaClass(Email.class);
    descriptor.addDirectQueryKey("id", "E_ID");
    descriptor.getInterfacePolicy().addParentInterface(Contact.class);
    descriptor.setTableName("INT_EML");
    descriptor.setPrimaryKeyFieldName("E_ID");
    descriptor.setSequenceNumberName("SEQ");
    descriptor.setSequenceNumberFieldName("E_ID");
    descriptor.addDirectMapping("emailID", "E_ID");
    descriptor.addDirectMapping("address", "ADDR");
    return descriptor;
Descriptor descriptor = new Descriptor();
    descriptor.setJavaClass(Phone.class);
    descriptor.getInterfacePolicy().addParentInterface(Contact.class);
    descriptor.addDirectQueryKey("id", "P_ID");
    descriptor.setTableName("INT_PHN");
    descriptor.setPrimaryKeyFieldName("P_ID");
    descriptor.setSequenceNumberName("SEQ");
    descriptor.setSequenceNumberFieldName("P_ID");
    descriptor.addDirectMapping("phoneID", "P_ID");
    descriptor.addDirectMapping("number", "P_NUM");
    return descriptor;
```

If the `Contact` interface extended another interface, you would call the following method to set its parent:

```
descriptor.getInterfacePolicy().addParentInterface(Interface.class);
```

Single Implementor Interfaces

Use single implementor interfaces for applications where only the domain objects' interface is visible. Each domain class has its own unique interface and no other domain class implements it. The references to other domain objects are also through interfaces.

In such applications, defining a descriptor for each interface would be expensive and may be unnecessary. TopLink does not force you to define descriptors for such interfaces. The descriptors are defined for the domain classes and the parent interface is set as usual.

During the initializing of a descriptor, the interface is given the descriptor of its implementor. This process allows queries on both the domain class and its interface. The only restriction is that each interface should have a unique implementor. In other words, a descriptor is not needed for an interface unless it has multiple implementors.

Implementing an Interface

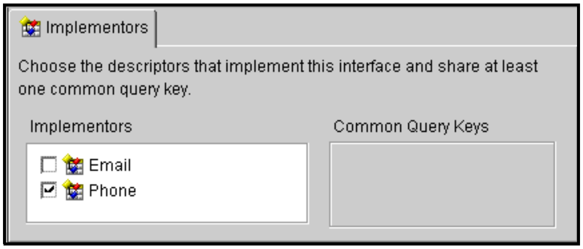
One-to-one mappings that reference interfaces that have multiple implementors are known as *variable* one-to-one mappings. For more information, see Chapter 6, "Understanding Relationship Mappings", and Chapter 4, "Understanding Descriptors".

Use this procedure to implement an interface.

To configure an interface descriptor

1. In the **Project Tree** pane, choose an interface.
2. On the **Implementors** tab in the **Properties** pane, choose the descriptors that implement this interface and share at least one common query key.

Figure 4–23 Implementors Tab



Implementors

Choose the descriptors that implement this interface and share at least one common query key.

Implementors

□ Email
☑ Phone

Common Query Keys

The **Common Query Keys** area displays all of the query keys for the interface's implementors.

To specify a class descriptor as a single implementor of an interface:

1. In the **Project Tree** pane, select the descriptor that will be the sole implementor of an interface.
2. If the **Interface Alias** advanced descriptor property is not visible for this descriptor, select **Set Advanced Properties > Interface Alias** from the **Selected** menu or the pop-up menu to create the **Interface Alias** page.
3. Select the interface that will serve as an alias for this descriptor on the **Interface Alias** page. This interface does not have to have a descriptor in the project, and in fact, if an associated descriptor exists, it will be removed. Every instance of the interface will now be treated as an instance of this class as well.

Working with Multiple Tables

Descriptors can use multiple tables in mappings. Use multiple tables when:

- A subclass is involved in inheritance, and its superclass is mapped to one table while the subclass has additional attributes that are mapped to a second table
- A class is not involved in inheritance and its data is spread out across multiple tables

When a descriptor has multiple tables, you must be able to join a row from the primary table to all of the additional tables. By default, TopLink assumes that the primary key of the first, or primary, table is included in the additional tables, thereby joining the tables.

TopLink also supports custom methods for joining tables.

Specifying Multi-table Info

Use the **Multi-table Info** tab to define multiple tables for a descriptor in the Mapping Workbench.

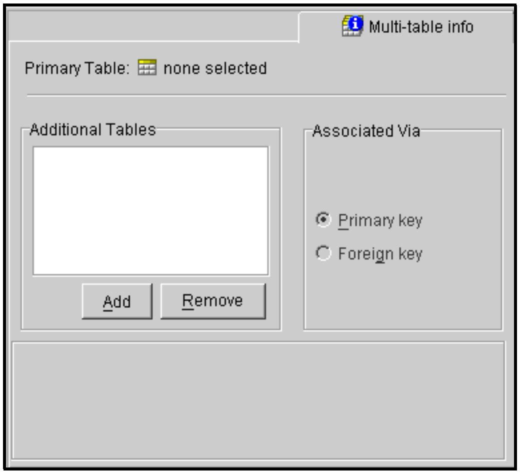
To associate multiple tables with a descriptor:

1. In the **Project Tree** pane, select a descriptor.

   If the **Multi-table Info** advanced property is not visible for the descriptor, right-click on the descriptor and choose **Set Advanced Properties > Multi-table Info** from pop-up menu or from the **Selected** menu.

2. Click on the **Multi-table info** tab.

Figure 4–24 Multi-table Info Tab



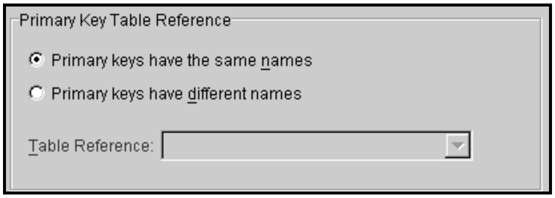
Primary Table: none selected

3. Use this table to enter data in each field:

Field	Description
Primary Table	The primary table for this descriptor. This field is for display only.
Additional Tables	Use the **Add** and **Remove** buttons to add or remove additional tables.
Associated Via	Specify if each **Additional Table** is associated by its **Primary** or **Foreign** key.

When associating a table via **Primary Key,** additional options appear on the Multi-table Info tab. Continue with "Primary Keys Match" on page 4-42 or "Primary Keys are Named Differently" on page 4-42 to assign the primary key.

Figure 4–25 Associating Multiple Tables via Primary Key



Primary Key Table Reference

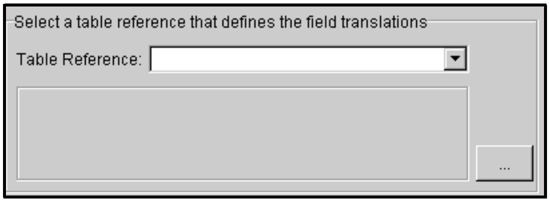
● Primary keys have the same names

○ Primary keys have different names

Table Reference:

When associating a table via **Foreign Key**, additional options (shown in Figure 4–25) appear. You must choose a reference that relates the relates the correct fields in the primary table to the primary keys in the selected table. Continue with "Tables are Related by Foreign Key in Primary Table" on page 4-43 to assign the foreign key.

Figure 4–26 Associating Multiple Tables via Foreign Key



Select a table reference that defines the field translations

Table Reference:

Primary Keys Match

When associating a descriptor with multiple tables in which the primary key field names are identically, you do not have to specify any additional information. Simply select the tables from the list of available tables on the **Multi-table Info** tab. The Mapping Workbench automatically selects the **Primary Keys Have the Same Names** option.

Primary Keys are Named Differently

If the primary keys of the additional table(s) are the same but named differently, you must specify how they relate to the primary key(s) of the default/primary table.

1. Select the associated table, and select **Associated Via Primary Key**.
2. Select **Primary Keys Have Different Names**.
3. In the **Primary Key Reference** area (Figure 4–25) Choose a table reference that relates how the primary keys of the primary table relate to the primary keys of the selected table. Use the drop-down list to select a primary key association.

Tables are Related by Foreign Key in Primary Table

If the primary keys of the additional table are not the same as the primary keys of the primary table, but are instead related to a different set of fields, you must set up a foreign key relation between the tables.

1. Select the associated table, and select **Associated Via Foreign Key**.
2. Use the drop-down list to select a foreign key reference that relates the correct fields in the primary table to the primary keys in the selected table. Click on the browse button to create a reference.

Working with a Copy Policy

The TopLink unit of work feature must be able to clone persistent objects. TopLink supports two ways of copying objects:

- By default, an object's default constructor is called to create a copy
- You may specify a method on the object to be used by TopLink to perform the copy, such as `clone`

Setting the Copy Policy

Use the **Copying** tab to specify how TopLink copies objects. TopLink supports the following methods:

- Using the object's default constructor to create a copy
- Specifying a method, such as `clone`

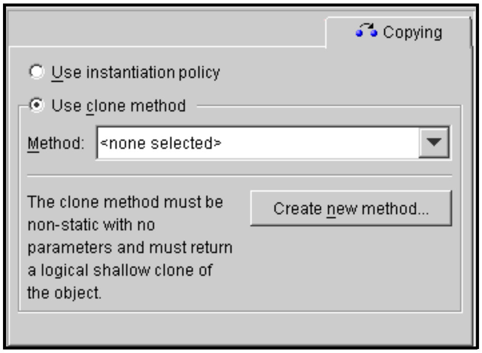
To specify a copy method:

1. Choose a descriptor in the **Project Tree** pane. Its properties appear in the **Properties** pane.

If the **Copying** advanced property is not visible for the descriptor, right-click on the descriptor and choose **Set Advanced Properties > Copying** from pop-up menu or from the **Selected** menu.

2. Click on the **Copying** tab in the **Properties** pane.

Figure 4–27 Copying Tab



○ Use instantiation policy

● Use clone method

Method: <none selected>

The clone method must be non-static with no parameters and must return a logical shallow clone of the object.

3. Use this table to enter data in each field:

Field	Description
Use instantiation policy	Specifies to create a new instance of the object using the descriptor's instantiation policy.
Use clone method	Specifies to calls the `clone()` method of the object.
Method	Select the clone method from the drop-list. Click on **Create New Method** to create a new method.

Working with Instantiation Policy

TopLink supports several ways to instantiate objects:

- By default, the default constructor of the class instantiates a new instance.
- If the application requires that objects be instantiated in other ways, the instantiation method can be customized.

You can use custom Java code to override the instantiation policy. Refer to *Oracle9iAS TopLink Foundation Library Guide* for details.

Setting Instantiation Policy

Use the **Instantiation** tab to specify if objects are instantiated by the default constructor, a specific method, or a factory.

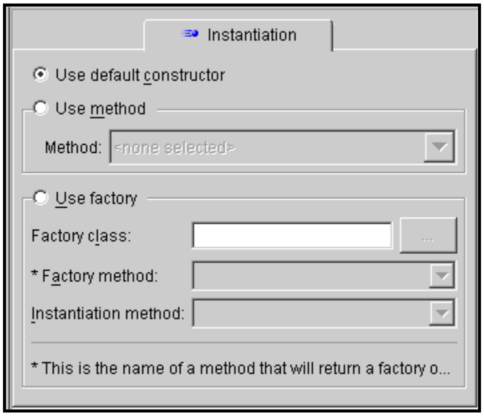
To set the instantiation policy:

1. In the **Project Tree** pane, select a descriptor.

   If the Instantiation advanced property is not visible for the descriptor, right-click on the descriptor and choose **Set Advanced Properties > Instantiation** from pop-up menu or from the **Selected** menu.

2. Click on the **Instantiation** tab.

Figure 4–28 Instantiation Tab



Instantiation

● Use default constructor

○ Use method

Method: <none selected>

○ Use factory

Factory class:

* Factory method:

Instantiation method:

* This is the name of a method that will return a factory o...

3. Use this table to enter data in each field:

Field	Description
Use Default Constructor	The default constructor of the class instantiates a new instance.
Use Method	Specify a **Method** to execute to create objects from the database.
Method	Name of a method to be executed to create objects from the database. The method must be a public, static method on the descriptor's class and must return a new instance of the object.
Use Factory	Refer to *Oracle9iAS TopLink Foundation Library Guide* for more information.

Field	Description
Factory Class	The class of the factory object that creates the new instances
Factory Method	The message to be sent to obtain a factory object
Instantiation Method	The method to be sent to the factory object to obtain a new instance that will be populated with data from the database

Working with a Wrapper Policy

TopLink allows you to use "wrappers" (or proxies) in cases where the persistent class is not the same class that is to be presented to users.

For example, in the Enterprise JavaBeans specification, the Entity bean class (the class that implements `javax.ejb.EntityBean`) is persistent but is hidden from users who interact with a class that implements `javax.ejb.EJBObject` (the "remote interface" class). In this example, the `EJBObject` acts as a proxy or wrapper for the `EntityBean`.

In cases where such a wrapper is used, TopLink continues to make the class specified in the descriptor persistent, but returns the appropriate instance of the wrapper whenever a persistent object is requested.

Use a "wrapper policy" to tell TopLink how to create wrappers for a particular persistent class, and how to obtain the underlying persistent object from a given wrapper instance.

If you specify a wrapper policy, TopLink uses the policy to "wrap" and "unwrap" persistent objects as required:

- Wrapper policies implement the interface `oracle.toplink.descriptors.WrapperPolicy`
- A wrapper policy is specified by setting the wrapper policy for the TopLink descriptor
- By default, no wrapper policy is used (the wrapper policy for a descriptor is null by default)
- Wrapper policies cannot be set using the Mapping Workbench, and can only be set using Java code

Note: Wrapper policies are advanced TopLink options. Using a wrapper policy may not be compatible with some Mapping Workbench features.

Setting the Wrapper Policy Using Java Code

The `Descriptor` class provides methods that are used in conjunction with the wrapper policy:

- `setWrapperPolicy(oracle.toplink.descriptors.WrapperPolicy)` can be invoked to provide a wrapper policy for the descriptor
- `getWrapperPolicy()` returns the wrapper policy for a descriptor

Refer to the *Oracle 9iAS TopLink Foundation Library Guide* for detailed information.

Working with Optimistic Locking

When using caching to provide performance benefits, you should also use a locking policy to manage database record modification in multi-user environments. Without a locking policy, it may be possible for users to see data stored in the cache that is no longer valid (sometimes called *stale* data).

Databases typically support the following locking policies:

- Optimistic – All users have read access to the data. When a user attempts to write a change, the application checks to ensure the data has not changed since the last read. TopLink supports multiple methods of optimistic locking.
- Pessimistic – The first user who access the data with the purpose of updating, locks the data until completing the update. TopLink supports pessimistic locking through `UnitOfWork` and `updateAndLockObject()`.
- No locking – The application does not verify that data is current.

Oracle recommends using optimistic locking to ensure that all users are working with valid data before committing changes. TopLink supports multiple locking policies for optimistic locking:

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

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

Using Version Locking Policies

For each of the following version locking policies, you must add a specific database field.

- For `VersionLockingPolicy`, add a numeric field
- For `TimestampLockingPolicy`, add a timestamp field

TopLink records the version as it reads an object from a table. When the client attempts to write the object, TopLink compares the version of the object with the version in the table record.

- If the versions are the same, the updated object is written to the version of both the table record and the object are
- If the versions are different, the write is disallowed because updated the object since this client initially

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

- For `VersionLockingPolicy`, the number in the version field increments by one
- For `TimestampLockingPolicy`, a new timestamp is inserted into the row (this policy can be configured to get the time from the or locally)

For both policies, the values of the write lock field can be the writable mapping within the object.

If the value is stored in the identity map, then by default an the version field. If the application does map the field, it must make mappings read-only to allow TopLink to control writing the fields.

Using Field Locking Policies

The following locking policies, included in TopLink, do not require any additional fields:

- `AllFieldsLockingPolicy`

- `ChangedFieldsLockingPolicy`
- `SelectedFieldsLockingPolicy`

All of these policies compare the current values of certain mapped previous values. When using these policies, a `UnitOfWork` must be used for updating the database. Each policy handles its specific way the policy.

- Whenever an object using `AllFieldsLockingPolicy` is updated or deleted, all the fields in that table are compared in the `where` clause. If any value in that table has been changed since the object was read, the update or delete fails.

> **Note:** This comparison is only on a per table basis. If an update is performed on an object that is mapped to multiple tables multiple table inheritance), only the changed table(s) appear in where clause.

- Whenever an object using `ChangedFieldsLockingPolicy` is updated, only the modified fields are compared. This allows for multiple clients to modify different parts of the same row without failure. Using this policy, a delete compares only on the primary
- Whenever an object using `SelectedFieldsLockingPolicy` is updated or deleted, a list of selected fields is compared in the statement. Updating these fields must be done by the application manually or though an event.

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

Specifying Advanced Optimistic Locking Policies

The TopLink optimistic locking policies (described in "Working with Optimistic Locking" on page 4-47) implement the `OptimisticLockingPolicy` interface, referenced throughout the TopLink code. You can create more policies by implementing this interface and implementing the methods defined.

Use the **Locking** tab (see Figure 4–16) to specify locking policies for the Mapping Workbench, or refer to the *Oracle 9iAS TopLink Foundation Library Guide* for more information.

Working with Identity Maps

TopLink uses *identity maps* to cache objects for performance and maintain object identity. The Mapping Workbench provides the following identity map types on the **Identity** tab (see Figure 4–13):

Table 4–2 Identity Maps

Identity Map	Description
Full identity map	Provides full caching and guaranteed identity. Caches all objects and does not remove them. This may be memory intensive when many objects are read. Do not use on batch type operations.
Soft cache weak identity map (default with JDK 1.2, available since JDK 1.2)	Similar to the weak identity map except that it maintains a most-frequently-used sub-cache. The size of the sub-cache is proportional to the size of the identity map as specified by descriptor's `setIdentityMapSize()` method. The sub-cache uses soft references to ensure that these objects are garbage-collected only if the system is low on memory.
Hard cache weak identity map (available since JDK 1.2)	Identical to the soft cache weak identity map except that it uses hard references in the sub-cache. This should be used if soft references do not behave properly on your platform.
Weak identity map (available since JDK 1.2)	Similar to the full identity map except that the map holds the objects using weak references. This allows for full garbage collection. It also provides full caching and guaranteed identity.
Cache identity map	Provides caching and identity, but does not guarantee identity. A cache identity map maintains a fixed number of objects specified by the application. Objects are removed from the cache on a least-recently-used basis. This method allows object identity for the most commonly used objects.
No identity map	Does not preserve object identity and does not cache objects.

Identity Map Size

The default identity map size is **100**.

- For the *cache* identity map policy, the size indicates the maximum number of objects stored in the identity map.
- For the *full* identify map policy, the size determines the starting size of the map.
- For the *soft/hard cache weak identity map*, the most-recently-used sub-cache is proportional to the size.

Design Guidelines

Use the following guidelines when using an identity map:

- If using a Java 2-compatible Virtual Machine (VM), objects with a long lifespan, and object identity is important, use a `SoftCacheWeakIdentityMap or HardCacheWeakIdentityMap` policy.
- If using a Java 2-compatible VM, and object identity is important and caching is not, use a `WeakIdentityMap` policy.
- If an object has a long lifespan or requires frequent access, or is important, use a `FullIdentityMap` policy.
- If an object has a short lifespan or requires frequent access, and identity is not important, use a `CacheIdentityMap` policy.
- If objects are discarded immediately after being read from the database, such as in a batch operation, use a `NoIdentityMap` policy

Note: The `NoIdentityMap` does not preserve object identity.

Using Object Identity

In a Java application, object identity is preserved if each object in memory is represented by one and only one object instance. Multiple retrievals of the same object return references to the same object instance – not multiple copies of the same object.

Maintaining object identity is extremely important when the application's object model contains circular references between objects. You must ensure that two are referencing each other directly, rather copies of each other. Object identity is important when multiple parts of the application may be modifying the same object simultaneously.

Identity can be turned off when object identity is not important (for example, for read-only objects).

Caching Objects

Identity maps maintain client-side object caches which boost by minimizing the number of database reads.

When the cache fills up, TopLink cleans up the cache based on the identity map policy.

Working with Query Keys

Use a *direct query key* as an alias for a field name. Query keys allow TopLink expressions to refer to a field using Java attribute names (such as `firstName`) rather than DBMS-specific names (such as `F_NAME`).

Use query keys to:

- Enhance code readability when defining TopLink expressions.
- Increase portability by making code independent of the database schema. If you rename a field, the query key could be redefined without changing any code that references it.
- Interface descriptors only define common query keys shared by implementors; the fields aliased could have different names in the implementor's tables

Automatically-generating Query Keys

TopLink automatically defines direct query keys for all direct mappings a special query key type for each mapping. Typically, use query keys to access fields that do not have direct mappings, such as the *version* field used for optimistic locking or the *type* field used for inheritance.

TopLink displays automatically generated query **Query Keys** tab of the **Properties** pane (see Figure 4–5). You cannot change these keys.

Example 4–4 Automatically Generated Query Key

For example, consider the `Employee` class in the TopLink tutorial: When you define a direct-to-field mapping from the `Employee` class, attribute `firstName` to the EMPLOYEE table, field F_NAME, you get a query key for free – it is automatically generated.

The following code example illustrates using an automatically-generated query key within the TopLink expression framework.

```
Vector employees = session.readAllObjects(Employee.class, new
ExpressionBuilder().get("firstName").equal("Bob"));
```

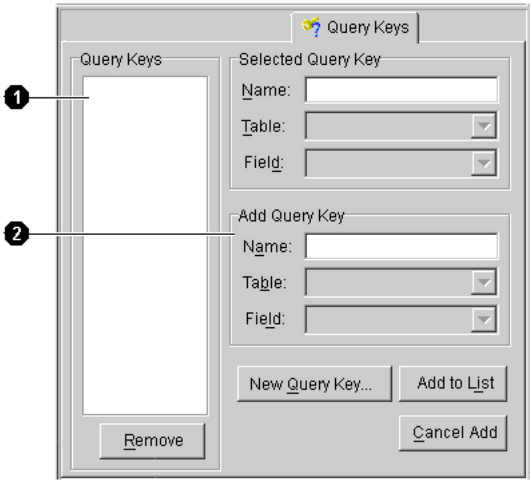
Creating a User-defined Query Key

In addition to the automatically generated query keys, you can define query keys for descriptors.

To use define a query key:

1. In the **Project Tree** pane, choose a descriptor.
2. In the **Properties** pane, click on the **Query Keys** tab.
3. Click on the **New Query Key** button.

Figure 4–29 The Add Query Key Window



User-interface items called out in Figure 4–29:

1. Existing query keys
2. New query key fields

4. Use this table to enter data in each field:

Field	Description
Name	Unique name of the query key. The **Name** must be different from any previously defined or automatically generated key.
Table	Table referenced by this query key.
Field	Field referenced by this query key.

5. Click the **Add to List** button. The new query key appears in the list in the **Query Keys** area of the tab.

Using Query Keys in Interface Descriptors

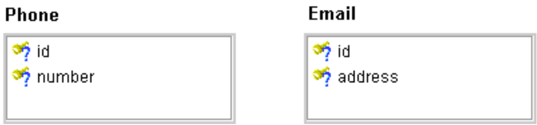
Interface descriptors are defined only with query keys that are shared among their implementors. In the descriptor for the interface, only the name of the query key is specified.

In each implementor descriptor, the key must be defined and with appropriate field from one of the implementor descriptor's tables.

This ensures that a single query key can be used to specify foreign key information from a descriptor that contains a mapping to the interface, even if the field names differ.

Consider an `Employee` that contains a contact, of type `Contact`. `Contact` is an interface with two implementors: `Phone` and `EmailAddress`. Each class has two attributes. The following figure illustrates the generated keys:

Figure 4–30 Auto-generated Query Keys for Phone and Email



Phone

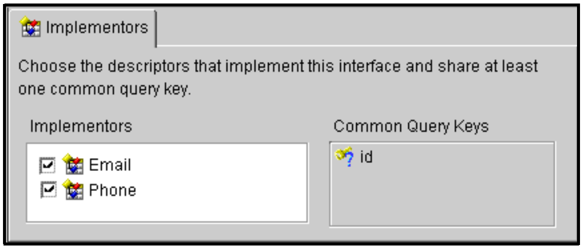
- id
- number

Email

- id
- address

Note: Both classes have an attribute, `id`, that is directly mapped to fields that have different names. However, a query key is generated for this attribute. For the `Contact` interface descriptor simply indicate that the `id` query key must be defined for each of the implementors, as shown in Figure 4–31.

Figure 4–31 Contact interface Descriptor with Common Query Key id



Implementors

Choose the descriptors that implement this interface and share at least one common query key.

Implementors

☑ Email
☑ Phone

Common Query Keys

id

Note: If either of the implementor classes did not have the id query key defined, that descriptor would be flagged as deficient.

Now that a descriptor with a commonly-shared query key has been defined for `Contact`, you can use it as the reference class for a variable one-to-one mapping. For example, you can now create a one-to-one mapping for the `contact` attribute of `Employee`. When you edit the foreign key field information for the mapping, you must match the `Employee` descriptor's tables to query keys from the `Contact` interface descriptor.

For more information see "Working with Interfaces" on page 4-36 and "Working with Relationship Mappings" on page 6-2.

Relationship Query Keys

TopLink supports query keys for relationship mappings and automatically defines them for all relationship mappings. You can use these keys to join across a relationship. One-to-one query keys define a joining relationship and are accessed through the `get()` method in expressions.

One-to-many and many-to-many query keys define a distinct join across a collection relationship and accessed through the `anyOf()` method in expressions. You can also define relationship query keys manually if `mapping` does not exist for the relationship. The relationship defined by the query key is data-level expressions.

Example 4–5 One-to-one Query Key

The following code example illustrates using a one-to-one query key within the TopLink expression framework

```
ExpressionBuilder employee = new ExpressionBuilder();
Vector employees = session.readAllObjects(Employee.class,
employee.get("address").get("city").equal("Ottawa"));
```

Defining Relationship Query Keys by Amending a Descriptor

Relationship query keys are not supported directly in the Mapping Workbench. To define a relationship query key, you must specify and write an amendment method. Register query keys by sending the `addQueryKey()` message.

Example 4–6 Defining One-to-one Query Key Example

The following code example illustrates how to define a one-to-one query key.

```
// Static amendment method in Address class, addresses do not know their owners
in the object-model, however you can still query on their owner if a
user-defined query key is defined
public static void addToDescriptor(Descriptor descriptor)
{
OneToOneQueryKey ownerQueryKey = new OneToOneQueryKey();
ownerQueryKey.setName("owner");
ownerQueryKey.setReferenceClass(Employee.class);
ExpressionBuilder builder = new ExpressionBuilder();
ownerQueryKey.setJoinCriteria(builder.getField("EMPLOYEE.ADDRESS_
ID").equal(builder.getParameter("ADDRESS.ADDRESS_ID")));
descriptor.addQueryKey(ownerQueryKey);
}
```

Working with Events

Use the event manager to specify specific events to occur whenever TopLink performs a read, update, delete, or insert on the database.

> **Note:** TopLink uses the Java event model.

Applications can receive descriptor events in the following ways:

- Implement `DescriptorEventListener` interface
- Subclass `DescriptorEventAdapter` adapter class
- Register an event method with a descriptor

Objects that implement the `DescriptorEventListener` interface can be registered with the descriptor event manager to be notified when any event occurs for that descriptor.

Alternately, you may wish to use the `DescriptorEventAdapter` class if your application does not require all of the methods defined in the interface. The `DescriptorEventAdapter` implements the `DescriptorEventListener` interface and defines an empty method for each method in the interface. To use the adapter, you must subclass it and then register your new object with the descriptor event manager.

Descriptor events can be used in many ways, including:

- Synchronizing the persistent objects with other systems, services and frameworks
- Maintaining non-persistent attributes of which TopLink is not aware
- Notifying other parts of the application when the persistent state of objects is changed
- Performing complex mappings or optimizations that are not directly supported by TopLink mappings

Use the descriptor's **Event** tab (see Figure 4–12) to specify events for a descriptor.

Example 4–7 Event

For example, if you want to invoke a method called `postBuild()` for an `Employee` object, the `postBuild()` method must be implemented in the `Employee` class. This method must also accept one parameter that is an instance of `DescriptorEvent` fully qualified with a package name.

Registering an Event with a Descriptor

A persistent class can register a public method as an event method. A descriptor calls the event method when a particular database operation occurs.

Event methods:

- Must be public so that TopLink can call them
- Must return void
- Must take a `DescriptorEvent` as a parameter

Example 4–8 Registering an Event

The following code illustrates an event method definition.

```
public void myEventHandler(DescriptorEvent event);
```

Supported Events

Events supported by the `DescriptorEventManager` include:

Post-X Methods:

- Post-Build — occurs after an object is built from the database.

- Post-Clone — occurs after an object has been cloned into a unit of work.
- Post-Merge — occurs after an object has been merged from a unit of work.
- Post-Refresh — occurs after an object is refreshed from the database.

Updating Methods:

- Pre-Update — occurs before an object is updated in the database. This may be called in a unit of work even if the object has no changes and does not require an update.
- About-to-Update — occurs when an object's row is updated in the database. This is called only if the object has changes in the unit of work.
- Post-Update — occurs after an object is updated in the database. This may be called in a unit of work even if the object has no changes and does not require an update.

Inserting Methods:

- Pre-Insert — occurs before an object is inserted in the database.
- About-to-Insert — occurs when an object's row is inserted in the database.
- Post-Insert — occurs after an object is inserted in the database.

Writing Methods:

- Pre-Write — occurs before an object is inserted or updated in the database. This occurs before preInsert/Update.
- Post-Write — occurs after an object is inserted or updated in the database. This occurs after preInsert/Update.

Deleting Methods:

- Pre-Delete — occurs before an object is deleted from the database.
- Post-Delete — occurs after an object is deleted from the database.

Working with Finders

In TopLink, use named queries to represent SQL or EJBQL finders to use in database accesses. You can create these finders within the mapping workbench.

When you create a finder for an EJB, the Mapping Workbench creates a named query and populates the descriptor alias with information from the `ejb-jar.xml` file.

Reserved finders are valid for projects with CMP persistence.

Working with Object-relational Descriptors

The object-relational paradigm extends traditional relational databases with object-oriented functionality. Oracle, IBM DB2, Informix and other DBMS databases allow users to store, access, and use complex data in more sophisticated ways.

The object-relational standard is an evolving standard and is mainly concerned with extending the database data structures and the SQL language (SQL 3).

The new features include:

- Structures or Object-types can be defined and stored on the database
- Collections/Arrays can be defined and stored on the database
- Structures/Object-types can have system-generated ObjectIDs
- Structures/Object-types can reference other structures through References or aggregation
- SQL 3, an extension to the SQL language that supports querying and manipulating the new object-types

Coinciding with object-relational changes, most database vendors are also extending their server architectures to support features such as:

- Embedded server-side Java Virtual Machines
- Java stored procedures
- CORBA, HTML and EJB support in the database

This section describes how the object-relational features affect TopLink descriptors and mappings. The server architecture changes are discussed in the *Oracle 9iAS TopLink: Foundation Library Guide*.

Effect on TopLink

Object-relational databases introduce several new features that allow more complex data to be stored and accessed. One advantage of object-relational databases is that the differences between the object model and data model can be reduced to the point that the two are almost identical. Although this makes the object-relational mapping process easier, it does not reduce the need for a persistence framework such as TopLink. Although the JDBC standard has been improved to take advantage of object-relational features in JDBC 2.0, it still remains a very low-level database interface. On top of JDBC, frameworks such as TopLink can provide applications with much more sophisticated functionality, including units of work, identity maps, expressions, querying, complex mappings, three-tier and enterprise application support.

TopLink provides object-relational support through a new type of descriptor object and several new types of mappings. See Chapter 7, "Understanding Object Relational Mappings" for more information.

Databases Supported

TopLink supports any JDBC 2.0 driver that complies with JDBC's 2.0 object-relational extensions. Contact your database and JDBC vendor to determine which object-relational features they support.

Defining Object-relational Descriptors

The TopLink Mapping Workbench does not currently support the object-relational descriptor and mappings. Support will be added to the Mapping Workbench in future releases.

You should be able to import most of the simple object-relational table structures into TopLink. Also, you can define the standard non-object-relational descriptor properties and mappings. You can use amendment methods to add any object-relational mappings and features to the descriptors.

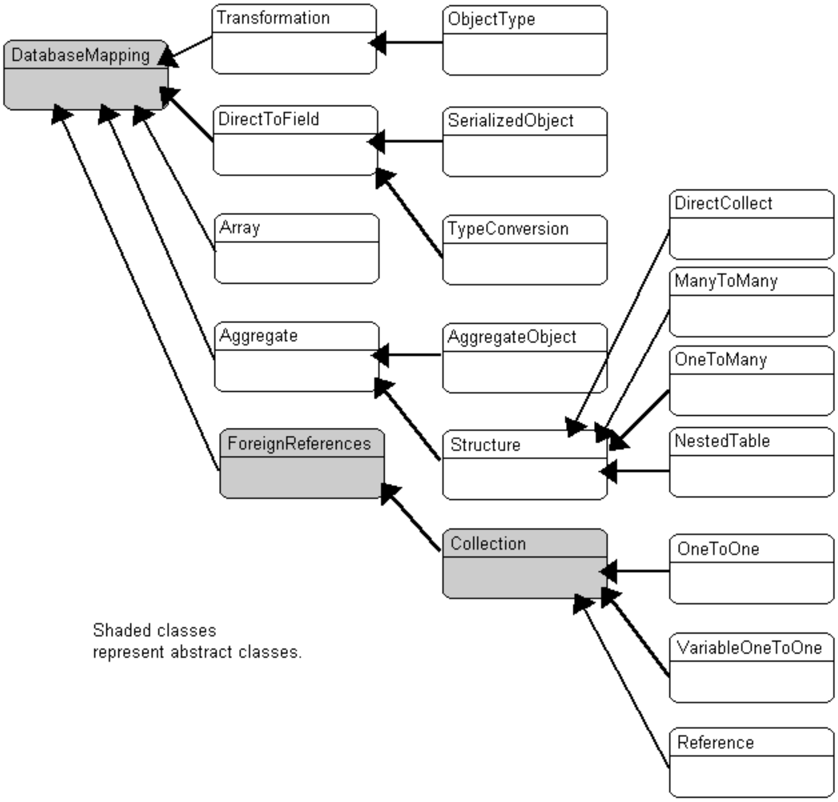
Working with Mappings

In TopLink, mappings define how an object's attributes are represented in the database.

- Direct mappings define how a persistent object refers to objects that do not have descriptors (for example, the JDK classes and primitives). See Chapter 5, "Understanding Direct Mappings" for details.
- Relationship mappings define how a persistent object refers to other persistent objects. See Chapter 6, "Understanding Relationship Mappings" and Chapter 7, "Understanding Object Relational Mappings" for details.

All of the mapping classes are derived from the `DatabaseMapping` class, as illustrated in Figure 4–32.

Figure 4–32 Mapping Classes Hierarchy

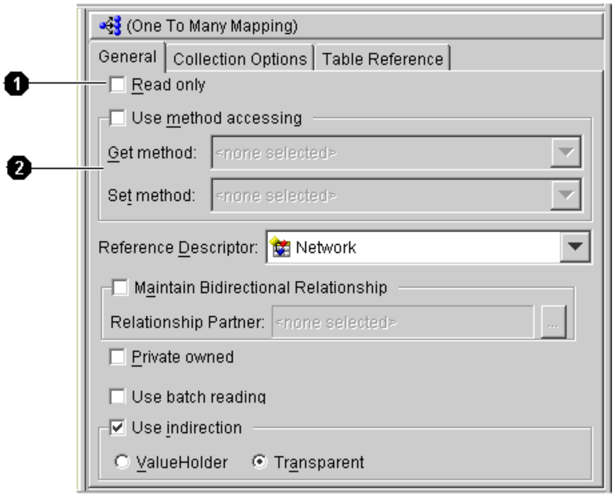


Working with Common Mapping Properties

TopLink associates each mapping with the attribute whose persistence it describes. To create a mapping in the Mapping Workbench, select the attribute to map from the **Project Tree** pane and then click on the appropriate button in the mapping toolbar (see Figure 1–5).

Use the mapping’s **Properties** pane to enter specific information for the mapping. Some mappings require more information that others and have multiple tabs in the **Properties** pane.

Figure 4–33 Sample Properties for a Mapping



□ Read only

□ Use method accessing

Get method: <none selected>

Set method: <none selected>

Reference Descriptor: Network

□ Maintain Bidirectional Relationship

Relationship Partner: <none selected>

□ Private owned

□ Use batch reading

☑ Use indirection

○ ValueHolder ● Transparent

Mapping properties called out in Figure 4–33:

1. Specify if read-only
2. Specify access method

Specifying Direct Access and Method Access

By default, TopLink uses direct access to access public attributes. Alternatively, you can use accessor methods to access object attributes when writing the attributes of the object to the database or reading the attributes of the object from the database. This is called method access.

The attribute’s visibility (**public**, **protected**, **private** or **package visibility**) and the supported version of JDK may restrict the type of access that you can use.

Starting in JDK 1.2, the Java Core Reflection API provides a means to suppress default Java language access control checks when using reflection. TopLink uses reflection to access the application’s persistent objects. This means that if you are using a VM that supports the API, then TopLink can access an attribute directly, regardless of its declared visibility.

Note: Private variable access under JDK 1.2 requires you to enable the security setting. Consult the JDK documentation for more information.

Oracle recommends using direct access whenever possible to improve performance and avoid executing any application-specific behavior while building objects.

Setting the Access Type

Use the **General** tab of the mapping **Properties** pane (see Figure 4–33) to set the access type as direct or method-based

To change the default access type used by all new mappings, use **Defaults** tab on the project **Properties** pane. See "Working with Default Properties" on page 2-7 for more information.

Note: If you change the access default, existing mappings retain their current access settings but new mappings will be created with the new default.

Specifying Read-only Settings

Use the **Read Only** check-box on the **General** tab of the mapping **Properties** pane (see Figure 4–33) to set a mapping to be read only. TopLink will not consider attributes associated with read-only mappings during update, and delete operations.

Because these operations are not actually performed for the mapping, any processes that are dependent on these operations (such as custom SQL or descriptor events) are not called for read-only. The attributes are still used for read operations.

Note: The primary key mappings must not be read-only.

Mappings defined for the write-lock or class indicator field *must* be read-only, unless the write-lock is configured not to be stored in the cache and the class indicator is part of the primary key.

Defaulting Null Values

Direct mappings include a `nullValue` attribute. Use this attribute to convert database null values to application-specific values (if application does not allow null values). This applies when typed as primitives. The null value must be set to the desired value, not the database value.

Null values translate in two directions: from null values read from the database to the specified value and from the specified value back to null when writing or querying. You can also use TopLink to set global default null values on a per-class basis. For more information, refer to the *Oracle 9iAS TopLink: Foundation Library Guide*.

Select the **Use Default Value when Database Field is Null** option on the **General** tab (see Figure 4–33) and the **Type** and **Value** drop-down lists to specify the null value.

> **Note:** You must specify the **Type** and **Value** in the mapping form.

Maintaining Bidirectional Relationships

Use the **Maintain Bidirectional Relationship Only** check-box on the **General** tab of the mapping **Properties** pane (see Figure 4–33) to maintain a bidirectional relationship for a one-to-one or one-to-many mapping. You can also specify the relationship partner.

Specifying Field Names and Multiple Tables

When defining mappings in code, TopLink assumes all mappings are in first table specified by the descriptor's `setTableName()` or `addTableName()` method. If the persistent class stores information in multiple tables, any messages sent that require field names should implemented to pass fully qualified names (that include the table name). Use the following syntax to fully qualify a field:

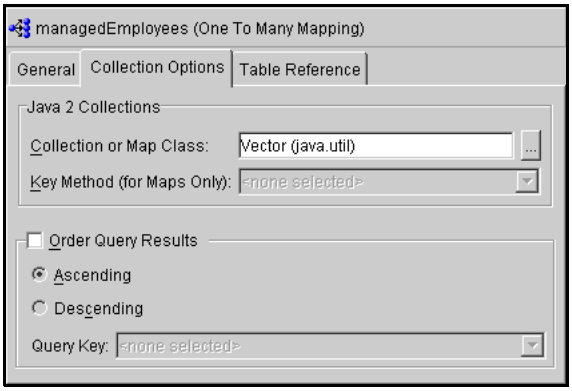
```
someMessage("tablename.fieldname");
```

Specifying Collection Properties

Some relationship mapping types (direct collection, one-to-many, and many-to-many) contain a **Collection Options** tab to allow you to specify collection options.

TopLink can populate a collection in ascending or descending order upon your specification. Query keys are automatically created for and with the same name as all attributes mapped as direct-to-field, type conversion, object type, and serialized object mappings.

Figure 4–34 Collection Options



Java 2 Collections

Collection or Map Class: Vector (java.util)

Key Method (for Maps Only): <none selected>

□ Order Query Results

● Ascending

○ Descending

Query Key: <none selected>

Use this table to enter data in each field:

Field	Description
Java 2 Collections:	
Collection or Map Class	Select the collection or map class to use for this collection mapping.
Key Method	
Order Query Results	Specify how the collection results are sorted for queries.

Specifying Mapping information in ejb-jar.xml File

For 2.0 CMP projects, the `ejb-jar.xml` files stores information on bean-to-bean relationships (i.e., mappings) in the `<relationship>` element. By updating this information in the `ejb-jar.xml`, the Mapping Workbench will create new mappings. You can then update the mapping information (such as reference tables).

If the information does not exist in the `ejb-jar.xml` file, you can build the mappings in the Mapping Workbench, then write the information to the file. See "Writing to the ejb-jar.xml File" on page 2-16 for more information.

5

Understanding Direct Mappings

In TopLink, direct mappings define how a persistent object refers to objects without descriptors, such as the JDK classes and primitives.

You can create the following direct mappings in TopLink:

- Direct-to-field mappings – Map a Java attribute directly to a database field (see "Working with Direct-to-field Mappings").
- Type conversion mappings – Map Java values with simple type conversions, such as from *character* to *string* (see "Working with Type Conversion Mappings").
- Object type mappings – Use an association to map values to the database (see "Working with Object Type Mappings").
- Serialized object mappings – Map serializable objects, such as multimedia objects, to database BLOB fields (see "Working with Serialized Object Mappings").
- Transformation mappings – Allow you to create custom mappings where one or more fields can be used to create the object be stored in the attribute (see "Working with Transformation Mappings").

Working with Direct Mappings

There are two basic ways of storing object attributes directly in a database table:

- The information can be stored directly if the attribute type is comparable to a database type.
- If there is no database primitive type that is logically comparable to the attribute's type, it must be transformed on its way to and from the database.

TopLink provides the following classes of direct mappings:

- Direct-to-field
- Type conversion
- Object type
- Transformation
- Serialized object

If the application's objects contain attributes that cannot be represented as direct-to-field, type conversion, or object-type mappings, the application must provide transformation routines for saving the attributes.

If a direct-to-field mapping cannot be used to perform the desired conversion, try type conversion and object type mappings before attempting to define a custom transformation mapping.

Working with Direct-to-field Mappings

Direct-to-field mappings map a Java attribute directly to a value database column. When the application writes a Java instance to database, it stores the value of the attribute in a field of the table column. TopLink supports the following types:

- **`java.lang`**: `Boolean`, `Float`, `Integer`, `String`, `Double`, `Long`, `Short`, `Byte`, `Byte[ ]`, `Character`, `Character[ ]`; all of the primitives associated with these classes
- **`java.math`**: `BigInteger`, `BigDecimal`
- **`java.sql`**: `Date`, `Time`, `Timestamp`
- **`java.util`**: `Date`, `Calendar`

While reading, direct-to-field mappings perform some simple one data conversions, as described in Table 5–1. You must use other direct mappings for two-way or more complete conversions.

Table 5–1 Type Conversions Provided by Direct-to-field Mappings

Java type	Database type
Integer, Float, Double, Byte, Short, BigDecimal, BigInteger, int, float, double, byte, short	NUMBER, NUMERIC, DECIMAL, FLOAT, DOUBLE, INT, SMALLINT, BIT, BOOLEAN

Table 5–1 Type Conversions Provided by Direct-to-field Mappings (Cont.)

Java type	Database type
Boolean, boolean	BOOLEAN, BIT, SMALLINT, NUMBER, NUMERIC, DECIMAL, FLOAT, DOUBLE, INT
String	VARCHAR, CHAR, VARCHAR2, CLOB, TEXT, LONG, LONG VARCHAR, MEMO
Character, char	CHAR
byte[ ]	BLOB, LONG RAW, IMAGE, RAW, VARBINARY, BINARY, LONG VARBINARY
Time	TIME
sql.Date	DATE (only applies to DB2)
Timestamp, util.Date, Calendar	TIMESTAMP (only applies to DB2)
sql.Date, Time, Timestamp, util.Date, Calendar	DATE, DATETIME (applies to Oracle, Sybase, SQL Server)

Direct-to-field mappings also allow you to specify a **null** value. This may be required if primitive types are used in the object and the database field allows null values.

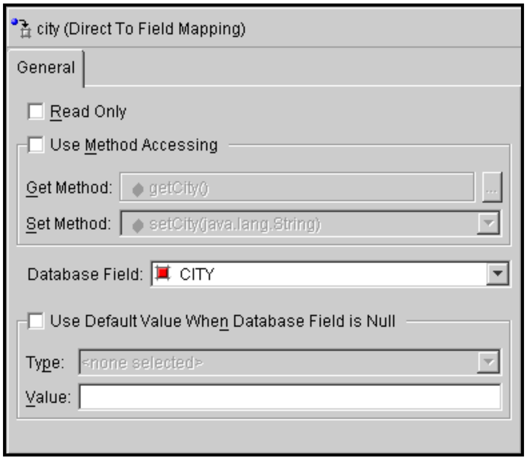
Creating Direct-to-field Mappings

Use this procedure to create a basic direct-to-field mapping to map a Java attribute directly to a value in a database.

To create a direct-to-field mapping:

1. Select the attribute to be mapped from the **Project Tree** pane.
2. Click the **Direct to Field Mapping** button from the mapping toolbar.
3. Use the **Database Field** drop-down list on the **General** tab on the **Properties** pane to select the appropriate database field.
4. Select the **Use Default Value When Database Field is Null** option to specify a default **Type** and **Value** to use if the database field is null.

Figure 5–1 Direct-to-field Mapping Properties



city (Direct To Field Mapping)

General

□ Read Only

□ Use Method Accessing

Get Method: getCity()

Set Method: setCity(java.lang.String)

Database Field: CITY

□ Use Default Value When Database Field is Null

Type: <none selected>

Value:

You can also specify:

- Read-only attributes – See "Specifying Read-only Settings" on page 4-63
- Access methods – See "Specifying Direct Access and Method Access" on page 4-62
- Null values – See "Defaulting Null Values" on page 4-64

Working with Type Conversion Mappings

Type conversion mappings explicitly map a database type to a Java type. For example, a `Number` in the database can be mapped to a `String` in Java, or a `java.util.Date` in Java can be mapped to a `java.sql.Date` in the database.

Creating Type Conversion Mappings

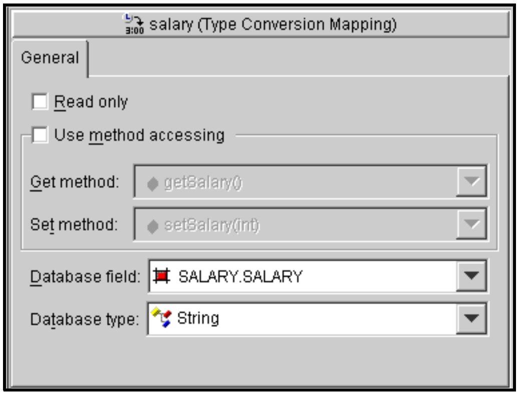
Use this procedure to create a type conversion mapping.

To create a type conversion mapping:

1. Select the attribute to be mapped from the **Project Tree** pane.
2. Click the **Type Conversion Mapping** button from the mapping toolbar.

3. Use the **Database field** and **Database type** drop-down lists on the **General** tab in the **Properties** pane to select the appropriate database field and database type.

Figure 5–2 Type Conversion Mapping Properties



salary (Type Conversion Mapping)

General

□ Read only

□ Use method accessing

Get method: getSalary()

Set method: setSalary(int)

Database field: SALARY.SALARY

Database type: String

You can also specify:

- Read-only attributes – See "Specifying Read-only Settings" on page 4-63
- Access methods – See "Specifying Direct Access and Method Access" on page 4-62

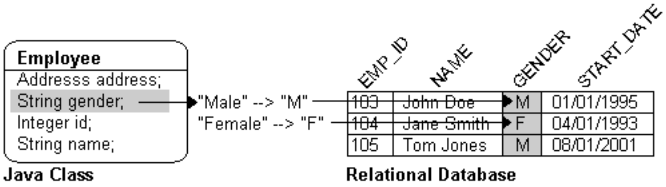
Working with Object Type Mappings

Object type mappings match a fixed number of database values to Java objects. Use these mappings when the values in the database differ from those in Java. Object types mappings are similar to direct-to-field mappings in all other respects.

Example 5–1 Object Type Mapping Example

The following figure illustrates an object type mapping between the `Employee` attribute `gender` and the relational database column `GENDER`. If the `gender` value in the Java class = `Male`, the system stores it in the `GENDER` database field as `M`; `Female` is stored as `F`.

Figure 5–3 Object Type Mappings



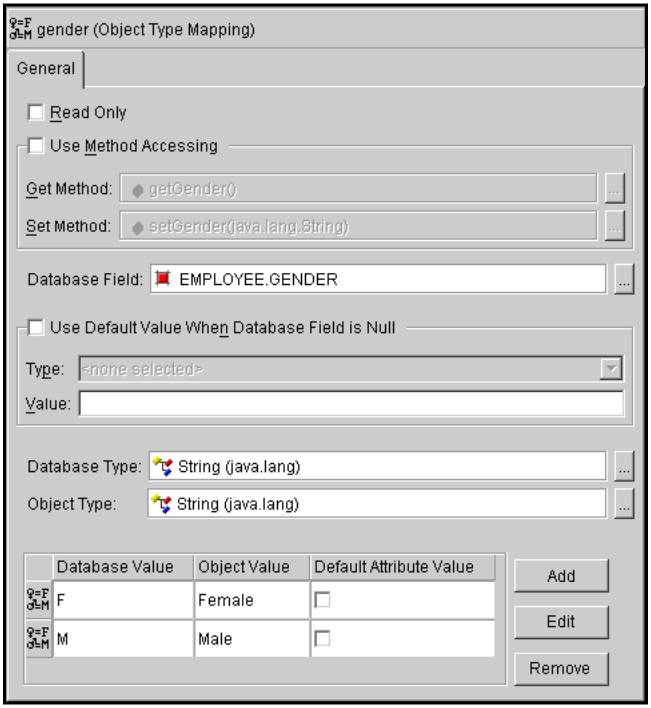
Creating Object Type Mappings

Use this procedure to create an object type mapping between an attribute and a database column.

To create a basic object type mapping:

1. In the **Project Tree** pane, choose the attribute to be mapped.
2. Click the **Object-Type Mapping** button from the mapping toolbar. The Object type mapping tab appears in the **Properties** pane.

Figure 5–4 Object Type Mapping General Properties



gender (Object Type Mapping)

□ Read Only

□ Use Method Accessing

Get Method: getGender()

Set Method: setGender(java.lang.String)

Database Field: EMPLOYEE.GENDER

□ Use Default Value When Database Field is Null

Type: <none selected>

Value:

Database Type: String (java.lang)

Object Type: String (java.lang)

Database Value	Object Value	Default Attribute Value
F	Female	□
M	Male	□

3. Choose the appropriate database field in the **Database Field** drop-down list.

4. Select **Use Default Value When Database Field is Null** to specify a default **Type** and **Value** to use if the database field is null.

5. Set the database type from the **Database Type** drop-down list and the Java type from the **Object type** drop-down list.

6. Click on **Add** to add **Database Value** and **Object Value** pairs to the table. Select the **Default Attribute Value** option for the value to use as the default.

   To remove a database value, select the value and click **Remove**.

You can also specify:

- Read-only attributes – See "Specifying Read-only Settings" on page 4-63
- Access methods – See "Specifying Direct Access and Method Access" on page 4-62

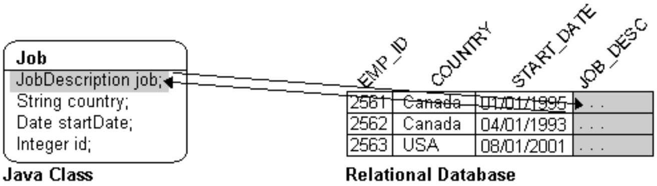
Working with Serialized Object Mappings

Serialized object mappings are used to store large data objects, such as multimedia files and BLOBs, in the database. Serialization transforms these large objects as a stream of bits.

Example 5–2 Serialized Object Mapping Example

Like direct-to-field mappings, serialized object mappings require an attribute and field to be specified, as illustrated in the following illustration.

Figure 5–5 Serialized Object Mappings



EMP_ID	COUNTRY	START_DATE	JOB_DESC
2561	Canada	01/01/1995	. . .
2562	Canada	04/01/1993	. . .
2563	USA	08/01/2001	. . .

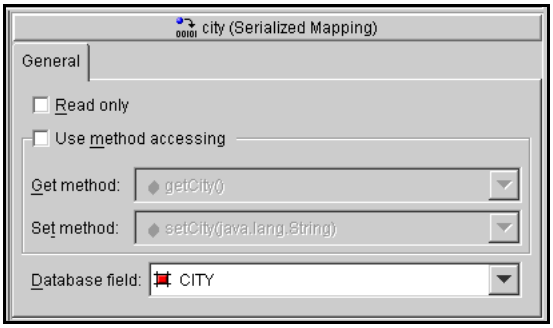
Creating Serialized Object Mappings

Use this procedure to create serialized object mappings.

To create a serialized object mapping:

1. In the **Project Tree** pane, choose the attribute to be mapped.
2. Click the **Serialized Mapping** button from the mapping toolbar.

Figure 5–6 Serialized Object Mapping Properties



city (Serialized Mapping)

General

☐ Read only

☐ Use method accessing

Get method: getCity()

Set method: setCity(java.lang.String)

Database field: CITY

3. Choose the appropriate database field in the **Database Field** drop-down list.

You can also specify:

- Read-only attributes – See "Specifying Read-only Settings" on page 4-63
- Access methods – See "Specifying Direct Access and Method Access" on page 4-62

Working with Transformation Mappings

Use transformation mappings for specialized translations between how a value is represented in Java and in the database.

> **Tip:** Use transformation mappings only when mapping multiple fields into a single attribute. Because of the complexity of transformation mappings, it is often easier to perform the transformation with `get/set` methods of a direct-to-field mapping.

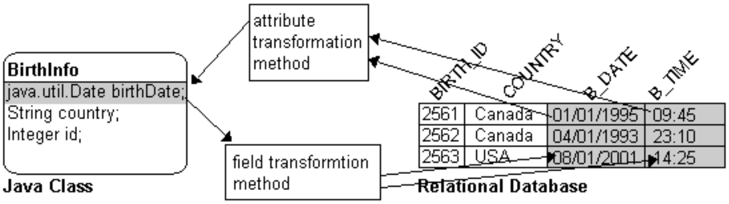
Often, a transformation mapping is appropriate when values from multiple fields are used to create an object. This type of mapping requires that you provide an *attribute transformation method* that is invoked when reading the object from the database. This method must have at least one parameter that is an instance of `DatabaseRow`. In your attribute transformation method, you can send the `get()` message to the `DatabaseRow` to get the value in a specific column. Your attribute transformation method may specify a second parameter, when is an instance of `Session`. The `Session` performs queries on the database to get additional values needed in the transformation. The method should *return* the value to be stored in the attribute.

Transformation mappings also require a *field transformation method* for each field to be written to the database when the object is saved. The transformation methods are specified in a dictionary associating each field with a method. The method returns the value to be stored in that field.

Example 5–3 Transformation Mapping Example

Figure 5–7 illustrates a transformation mapping. The values from the B_DATE and B_TIME fields are used to create a `java.util.Date` to be stored in the `birthDate` attribute.

Figure 5–7 Transformation Mappings



BIRTH_ID	COUNTRY	B_DATE	B_TIME
2561	Canada	01/01/1995	09:45
2562	Canada	04/01/1993	23:10
2563	USA	08/01/2001	14:25

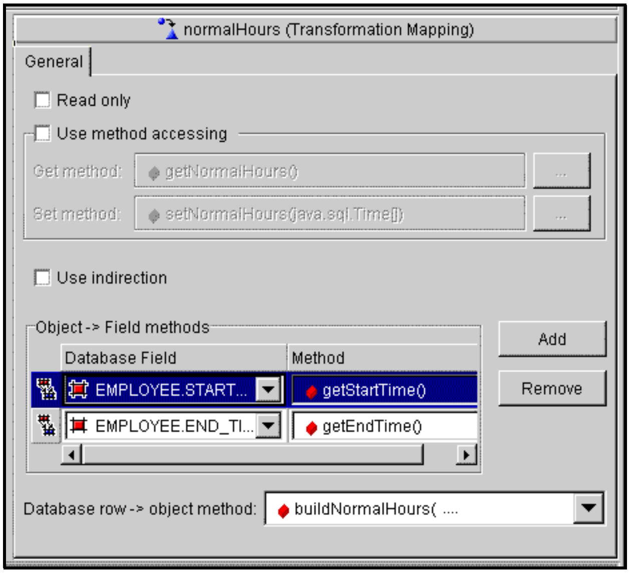
Creating Transformation Mappings

Use this procedure to create transformation mappings in the Mapping Workbench.

To create a transformation mapping:

1. In the **Project Tree** pane choose the attribute to be mapped.
2. Click the **Transformation Mapping** button from the **Mapping** toolbar.

Figure 5–8 Transformation Mapping Tab



normalHours (Transformation Mapping)

General

□ Read only

□ Use method accessing

Get method: getNormalHours()

Set method: setNormalHours(java.sql.Time[])

□ Use indirection

Object -> Field methods

Database Field	Method
EMPLOYEE.START...	getStartTime()
EMPLOYEE.END_TI...	getEndTime()

Database row -> object method: buildNormalHours(....

3. Use the **Database Row --> Object Method** drop-down list to select a method to convert the database row into an object.

> **Note:** The method must have parameter `(DatabaseRow)` or parameters `(DatabaseRow, Session)`.

4. Click on **Add** to add field transformation methods to the descriptor.

   To remove a transformation method, select the method and click on **Remove**.

5. Use the **Use indirection** check box to specify if the creation of the target object requires extensive computational resources. If selected, TopLink uses indirection objects. See "Working with Indirection" on page 6-5 for more information.

6. After specifying the details of the mapping, create the attribute field transformation methods in the associated Java class (see Example 5–4, "Transformation Mapping Code Example").

You can also specify:

- Read-only attributes – See "Specifying Read-only Settings" on page 4-63
- Access methods – See "Specifying Direct Access and Method Access" on page 4-62

Example 5–4 Transformation Mapping Code Example

The following code example illustrates the methods required for a transformation mapping.

```
// Get method for the normalHours attribute since method access indicated
access public Time[] getNormalHours()
{
    return normalHours;
}
// Set method for the normalHours attribute since method access indicated
access public void setNormalHours(Time[] theNormalHours)
{
    normalHours = theNormalHours;
}
// Create attribute transformation method to read from the database row
//** Builds the normalHours Vector. IMPORTANT: This method builds the value but
does not set it. The mapping will set it using method or direct access as
defined in the descriptor. */
public Time[] getNormalHoursFromRow(DatabaseRow row)
{
    Time[] hours = new Time[2];
    hours[0] = (Time)row.get("START_TIME");
    hours[1] = (Time)row.get("END_TIME");
    return hours;
}
// Define a field transformation method to write out the start time. Return the
first element of the normalHours attribute.
public java.sql.Time getStartTime()
{
    return getNormalHours()[0];
}
// Define a field transformation method to write out the end time. Return the
last element of the normalHours attribute.
public java.sql.Time getEndTime()
{
    return getNormalHours()[1];
}
```

Specifying Advanced Features Available by Amending the Descriptor

In TopLink, transformation mappings do not require you to specify an attribute.

A field may be mapped from a computed value that does not map to a logical attribute. This, in effect, constitutes a write-only mapping. In the Mapping Workbench, all mappings are associated with an attribute before any other information can be specified. Therefore, to use a write-only mapping, you must build it by amending the descriptor. The mapping itself has no attribute name, `get` and `set` methods, or attribute method. In your amendment method, simply create an instance of `TransformationMapping` and send `addFieldTransformation()` message for each field to be written.

Example 5–5 Descriptor Amendment Examples

The following code example illustrates creating a write-only transformation mapping and adding it to the descriptor.

```
public static void addToDescriptor(Descriptor descriptor) {
// Create a Transformation mapping and add it to the descriptor.
TransformationMapping transMapping = new
transMapping.addFieldTransformation(“WRITE_DATE”,
“descriptor.addMapping(transMapping);
}
```

The following example illustrates how to create a one-way transformation mapping by using the inheritance indicator field of the primary key. Map the class as a normal, including the other part of the primary key, and the inheritance through the type field.

Note: The Mapping Workbench will display a neediness error because the class indicator field part of the primary key is not mapped. Use the following code to create an amendment method to map the indicator field.

Create an amendment method for the class:

```
public void addToDescriptor(Descriptor descriptor) {
    TransformationMapping keyMapping = new TransformationMapping();
    keyMapping.addFieldTranslation(“PROJECT.PROJ_TYPE”, “getType”);
    descriptor.addMapping(keyMapping);}
```

Define the `getType` method on the class to return its type value:

```
Project>>public abstract String getType();
LargeProject>>public String getType() { return “L”; }
SmallProject>>public String getType() { return “S”; }
```

Refer to "Amending Descriptors After Loading" on page 4-18 for more information.

6

Understanding Relationship Mappings

Relational mappings define how persistent objects reference other persistent objects. TopLink provides the following relationship mappings:

- Direct collection mappings – Map Java collections of objects that do not have descriptors (see "Working with Direct Collection Mappings").
- Aggregate object mappings – Strict one-to-one mappings that require both objects to exist in the same database row (see "Working with Aggregate Object Mappings").
- One-to-one mappings – Map a reference to another persistent Java object to the database (see "Working with One-to-one Mappings").
- Variable one-to-one mappings – Map a reference to an interface to the database (see "Working with Variable One-to-one Mappings").
- One-to-many mappings – Map Java collections of persistent objects to the database (see "Working with One-to-many Mappings").
- Aggregate collection mappings also map Java collections of persistent objects to the database (see "Working with Aggregate Collection Mappings").
- Many-to-many mappings use an association table to map Java collections of persistent objects to the database (see "Working with Many-to-many Mappings").

TopLink also provides object-relational relationship mappings (see Chapter 5, "Understanding Direct Mappings" and Chapter 7, "Understanding Object Relational Mappings").

All TopLink relationship mappings are uni-directional, from the class being described (the *source* class) to the class with which it is associated (the *target* class). The target class does not have a reference to the source class in a uni-directional relationship.

To implement a bi-directional relationship (classes that reference each other) use two unidirectional mappings with the sources and targets reversed.

Working with Relationship Mappings

Persistent objects use relationship mappings to store references to instances of other persistent classes. The appropriate mapping class is chosen primarily by the cardinality of the relationship.

Specifying Private or Independent Relationships

In TopLink, object relationships can be either private or independent.

- In a private relationship the target object is a private component of the source object. The target object cannot exist without the source and is accessible only via the source object. Destroying the source object will also destroy the target object.
- In an independent relationship the source and target are public objects that exist independently. Destroying one object does not necessarily imply the destruction of the other.

Aggregate object mappings are private by default, since the target object shares the same row as the source object. One-to-one, one-to-many, and many-to-many mappings can be independent or private, depending upon the application. Normally, many-to-many mappings are independent by definition; however, because a many-to-many mapping can be used to implement a logical one-to-many without requiring a back reference in the target to the source, TopLink allows many-to-many mappings to be private as well as independent.

Tip: TopLink automatically supports private relationships. Whenever an object is written to the database, any private objects it owns are also written to the database. When an object is removed from the database, any private objects it owns are also removed. You should be aware of this when creating new systems, since it may affect both the behavior and the performance of your application.

Working with Foreign Keys

TopLink uses *references* to maintain foreign key information. TopLink defines the reference as a property of the table containing the foreign key. This may or may not correspond to an actual constraint that exists on the database.

If you import tables from the database, TopLink creates references that correspond to existing database constraints (if the driver supports this). You can also define any number of references in the Mapping Workbench without creating similar constraints on the database.

TopLink uses these references when defining relationship mappings and descriptors' multiple table associations.

Understanding Foreign Keys

A foreign key is a combination of columns that reference a unique key, usually the primary key, in another table. Foreign keys can be any number of fields (similar to primary key), all of which are treated as a unit. A foreign key and the parent key it references must have the same number and type of fields.

Relationship mappings use foreign keys to find information in the database so that the target object(s) can be instantiated. For example, if every `Employee` has an attribute address that contains an instance of `Address` (which has its own descriptor and table), the one-to-one mapping for the address attribute would specify foreign key information to find an address for a particular `Employee`.

TopLink classifies foreign keys into two categories in mappings — **foreign keys** and **target foreign keys**:

> **Caution:** Make sure you fully understand the distinction between *foreign key* and *target foreign key* before defining a mapping.

- In a *foreign key* the key is found in the table associated with the mapping's own descriptor. In the previous example, a foreign key to `ADDRESS` would be in the `EMPLOYEE` table.
- In a target foreign key the reference is from the target object's table back to the key from the mapping's descriptor's table. In the previous example the `ADDRESS` table would have a foreign key to `EMPLOYEE`.

Specifying Foreign Keys

If you import tables from the database, TopLink creates references that correspond to existing database constraints (if supported by the driver). You can also define references in TopLink without creating similar constraints on the database.

To display existing references for a table, use the **References** tab. References that contain the **On Database** option will create a constraint that corresponds to the references.

> **Note:** Your database driver must support this.

To create a foreign key:

1. Choose a database table in the **Project Tree** pane that will contain the foreign key.
2. Click on the **References** tab in the **Properties** pane.
3. Select a reference table. See "Creating table references" on page 3-10 for more information.
4. Add a key pair for the reference. See "Creating Field References" on page 3-11 for more information.

   Use the **Source Field** and **Target Field** drop-down lists to select the appropriate fields on the source and target tables.

   Repeat step 4 for each foreign key field.

Working with a Container Policy

A container policy specifies the concrete class TopLink should use when reading target objects from the database. You can specify a container policy for collection mappings (`DirectCollectionMapping`, `OneToManyMapping`, and `ManyToManyMapping`) and for read-all queries (`ReadAllQuery`).

Starting with JDK 1.2 the collection mappings can use any concrete class that implements either the `java.util.Collection` interface or the `java.util.Map interface`.

When using TopLink with JDK 1.2 (or later), you can map object attributes declared as `Collection` or `Map`, or any sub-interface of these two interfaces, or as a class that implements one of these two interfaces. You must specify in the mapping the concrete container class to be used. When TopLink reads objects from the database that contain an attribute mapped with a collection mapping, the attribute is set with an instance of the concrete class specified. By default, a collection mapping's container class is `java.util.Vector`.

Read-all queries also require a container policy to specify how the result objects are to be returned. The default container is `java.util.Vector`.

Container policies cannot be used to specify a custom container class when using indirect containers.

Overriding the Default Container Policy

For collection mappings, you can specify the container class in the Mapping Workbench (see "Working with Direct Collection Mappings" on page 6-28).

To set the container policy without using the Mapping Workbench, the following API is available for both `CollectionMapping` and `ReadAllQuery`:

- `useCollectionClass(Class)` – Specifies the concrete `Collection` class to use as a container for the objects in the collection. In JDK 1.2, the class must implement the `java.util.Collection` interface.
- `useMapClass(Class, String)` – Specifies the concrete Map class to use as a container for the objects in the collection. In JDK 1.2 the class must implement the `java.util.Map interface`.

  Also specified is the name of the zero argument method whose result, when called on the target object, is used as the key in the `Hashtable` or `Map`. This method must return an object that is a valid key in the `Hashtable` or `Map`.

Working with Indirection

Using indirection objects may improve the performance of TopLink object relationships. An indirection object takes the place of an application object so that the application object is not read from the database until it is needed.

Without indirection, when TopLink retrieves a persistent object, it also retrieves all the objects referenced by that object. This may result in lower performance for some applications. Using indirection allows TopLink to create "stand-ins" for related objects, resulting in significant performance improvements, especially when the application is only interested in the contents of the retrieved object rather than the objects to which it is related.

Understanding Indirection

Indirection is available for transformation mappings and for direct collection, one-to-one, one-to-many, and many-to-many relationship mappings.

You can enable or disable indirection for each mapping individually. By default, indirection is enabled for relationship mappings and disabled for transformation mappings. Indirection should only be enabled for transformation mappings if the execution of the transformation method is a resource-intensive task, such as accessing the database.

- *Indirection disabled*: An indirection object is not used. Whenever an object is retrieved from the database, all of the objects associated with it through the mapping are also read.
- *Indirection enabled*: A value holder is used to represent the entire relationship. When an object is retrieved from the database, a value holder is created and stored in the attribute corresponding to the mapping. The first time the value holder is accessed, it retrieves the **related object from the database**.

In addition to this standard version of indirection, collection mappings (direct collection, one-to-many, and many-to-many) can use indirect **containers**.

Using Value Holder Indirection

Persistent classes that use indirection must replace relationship attributes with value holder attributes. A value holder is an instance of a class that implements the `ValueHolderInterface` interface, such as `ValueHolder`. This object stores the information necessary to retrieve the object it is replacing from the database. If the application does not access the value holder, the replaced object is never read from the database.

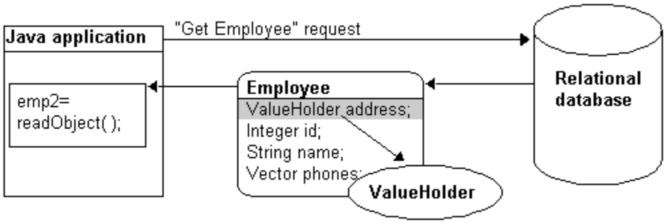
When using method access, the `get` and `set` methods specified for the mapping must access an instance of `ValueHolderInterface` rather than the object referenced by the value holder.

To obtain the object that the value holder replaces, use the `getValue()` and `setValue()` methods of the `ValueHolderInterface` class. A convenient way of using these methods is to hide the `getValue` and `setValue` methods of the `ValueHolderInterface` inside `get` and `set` methods, as in the following example.

Example 6–1 Value Holder Indirection Example

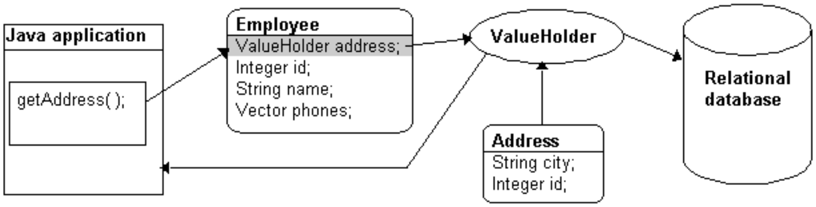
The following figure illustrates the `Employee` object being read from the database. The `Address` object is not read and will not be created unless it is accessed.

Figure 6–1 Address Object Not Read



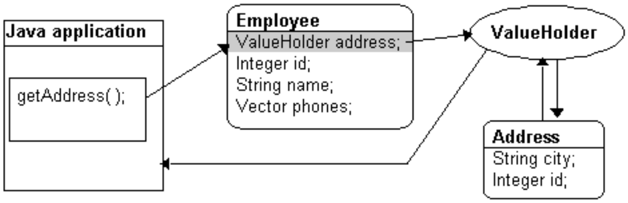
The first time the address is accessed, as in the following figure, the `ValueHolder` reads and returns the `Address` object.

Figure 6–2 Initial Request



Subsequent requests for the address do not access the database, as shown in the following figure.

Figure 6–3 Subsequent Requests



Specifying Indirection

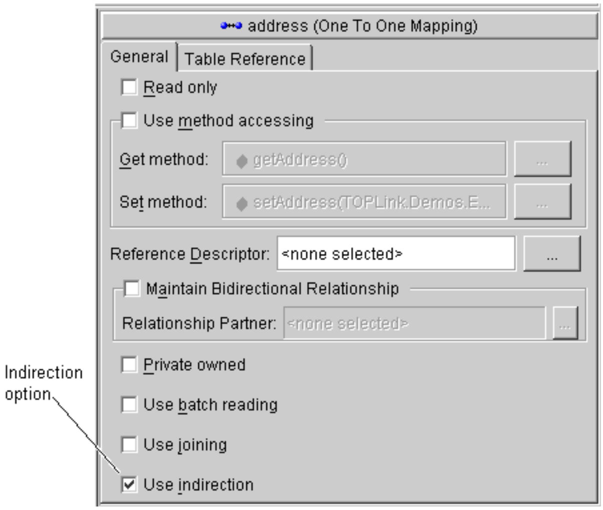
Use this procedure to specify that a mapping uses indirection.

To specify indirection:

1. In the **Project Tree** pane, select the mapping to be mapped and click the appropriate button from the mapping toolbar.

   The window appears in the **Properties** pane.

Figure 6–4 Sample Mapping Properties



address (One To One Mapping)

General | Table Reference

☐ Read only

☐ Use method accessing

Get method: getAddress() ...

Set method: setAddress(TOPLink.Demos.E... ...

Reference Descriptor: <none selected> ...

☐ Maintain Bidirectional Relationship

Relationship Partner: <none selected> ...

☐ Private owned

☐ Use batch reading

☐ Use joining

☑ Use indirection

Indirection option

2. On the **General** tab, select **Use Indirection** to specify that the mapping uses indirection.

Changing Java Classes to Use Indirection

Attributes using indirection must conform to the `ValueHolderInterface`. You can change your attribute types in the Class Editor without re-importing your Java classes. Ensure that you change the attribute types in your Java code as well. Attributes that are typed incorrectly will be marked as deficient.

In addition to changing the attribute's type, you may also need to change its accessor methods. If you use method access, TopLink requires accessors to the *indirection* object itself, so your get method returns an instance that conforms to `ValueHolderInterface` and your set method accepts one argument that conforms to the same. If the instance variable returns a Vector instead of an object then the value holder should be defined in the constructor as follows:

```
addresses = new ValueHolder(new Vector());
```

In any case, the application uses the `getAddress()` and `setAddress()` methods to access the `Address` object. With indirection, TopLink uses the `getAddressHolder()` and `setAddressHolder()` methods when saving and retrieving instances to and from the database.

Refer to the *TopLink: Foundation Library Guide* for details.

Example 6–2 Indirection Example

The following code illustrates the `Employee` class using indirection with method access for a one-to-one mapping to `Address`.

The class definition is modified so that the address attribute of `Employee` is a `ValueHolderInterface` instead of an `Address` and appropriate get and set methods are supplied.

```
// Initialize ValueHolders in Employee Constructor
public Employee() {
address = new ValueHolder();
}
protected ValueHolderInterface address;

// 'Get' and 'Set' accessor methods registered with the mapping and used by
Oracle 9iAS TopLink.
public ValueHolderInterface getAddressHolder() {
return address;
}
public void setAddressHolder(ValueHolderInterface holder) {
address = holder;
}

// 'Get' and 'Set' accessor methods used by the application to access the
attribute.
public Address getAddress() {
return (Address) address.getValue();
}
public void setAddress(Address theAddress) {
address.setValue(theAddress);
}
```

Working with Transparent Indirection

Transparent indirection allows you to declare any relationship attribute of a persistent class that holds a collection of related objects as a java.util.Collection, java.util.Map, java.util.Vector, or java.util.Hastable. TopLink will use an indirection object that implements the appropriate interface and also performs "just in time" reading of the related objects. When using transparent indirection, you do not have to declare the attributes as ValueHolderInterface.

You can specify transparent indirection from the Mapping Workbench. Newly created collection mappings use transparent indirection by default, if their attribute *is not* a ValueHolderInterface.

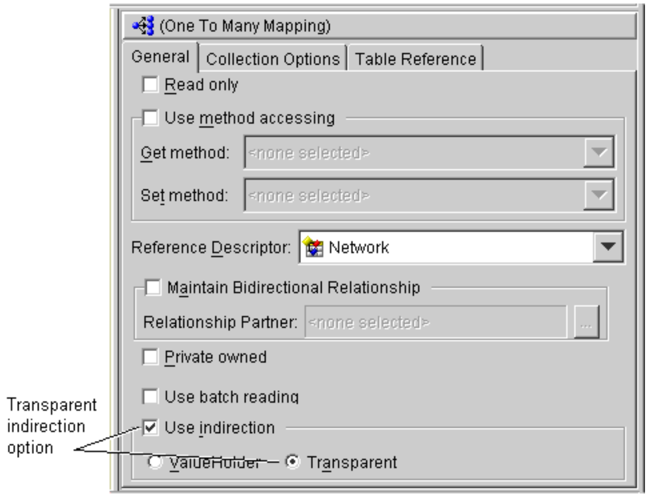
Specifying Transparent Indirection

Use this procedure to use transparent indirection.

Using Transparent Indirection:

1. In the **Project Tree** pane, select the attribute. The mapping window appears in the **Properties** pane.

Figure 6–5 Sample Mapping properties



(One To Many Mapping)

General | Collection Options | Table Reference

□ Read only

□ Use method accessing

Get method: <none selected>

Set method: <none selected>

Reference Descriptor: Network

□ Maintain Bidirectional Relationship

Relationship Partner: <none selected>

□ Private owned

□ Use batch reading

☑ Use indirection

○ ValueHolder ● Transparent

Transparent indirection option

2. On the **General** tab, select the **Use Indirection** option for attributes that use indirection.

3. Select the **Transparent** indirection option.

Working with Proxy Indirection

Introduced in JDK 1.3, the Java class Proxy allows you to use dynamic proxy objects as stand-ins for a defined interface. Certain TopLink mappings (OneToOneMapping, VariableOneToOneMapping, ReferenceMapping, and TransformationMapping) can be configured to use proxy indirection which gives you the benefits of TopLink indirection without the need to include TopLink classes in your domain model. Basically, proxy indirection is to one-to-one relationship mappings as indirect containers are to collection mappings.

Although the Mapping Workbench does not support proxy indirection, you can use the useProxyIndirection method in an amendment method.

To use proxy indirection, your domain model must satisfy the following criteria:

- The target class of the one-to-one relationship must implement a public interface
- The one-to-one attribute on the source class must be typed as that interface
- If method accessing is used, then the get() and set() methods must use the interface

Example 6–3 Proxy indirection Examples

The following code illustrates an Employee->Address one-to-one relationship.

```
public interface Employee {
    public String getName();
    public Address getAddress();
    public void setName(String value);
    public void setAddress(Address value);
    . . .
}
public class EmployeeImpl implements Employee {
    public String name;
    public Address address;
    . . .
    public Address getAddress() {
        return this.address;
    }
    public void setAddress(Address value) {
        this.address = value;
    }
}
public interface Address {
    public String getStreet();
    public void setStreet(String value);
    . . .
}
public class AddressImpl implements Address {
    public String street;
    . . .
}
```

In this example, both the `EmployeeImpl` and the `AddressImpl` classes implement public interfaces (`Employee` and `Address` respectively). Therefore, because the `AddressImpl` is the target of the one-to-one relationship, it is the only class that must implement an interface. However, if the `EmployeeImpl` is ever to be the target of another one-to-one relationship using transparent indirection, it must also implement an interface.

The following code illustrates this relationship using proxy indirection.

```
Employee emp = (Employee) session.readObject(Employee.class);
System.out.println(emp.toString());
System.out.println(emp.getAddress().toString());
// Would print:
[Employee] John Smith
{ IndirectProxy: not instantiated }
String street = emp.getAddress().getStreet();
// Triggers database read to get Address information
System.out.println(emp.toString());
System.out.println(emp.getAddress().toString());
// Would print:
[Employee] John Smith
{ [Address] 123 Main St. }
```

Using proxy indirection does not change how you instantiate your own domain objects for insert. You still use the following code:

```
Employee emp = new EmployeeImpl("John Smith");
Address add = new AddressImpl("123 Main St.");
emp.setAddress(add);
```

Implementing Proxy Indirection in Java

To enable proxy indirection in Java code, use the following API for `ObjectReferenceMapping`:

- `useProxyIndirection()` – indicates that TopLink should use proxy indirection for this mapping. When the source object is read from the database, a proxy for the target object is created and used in place of the "real" target object. When any method other than `g-string()` is called on the proxy, the "real" data will be read from the database.

Example 6–4 Proxy indirection Example

The following code example illustrates using proxy indirection.

```
// Define the 1:1 mapping, and specify that Proxy Indirection should be used
OneToOneMapping addressMapping = new OneToOneMapping();
addressMapping.setAttributeName("address");
addressMapping.setReferenceClass(AddressImpl.class);
addressMapping.setForeignKeyFieldName("ADDRESS_ID");
addressMapping.setSetMethodName("setAddress");
addressMapping.setGetMethodName("getAddress");
addressMapping.useProxyIndirection();
descriptor.addMapping(addressMapping);
. . .
```

Optimizing for Queries

You can configure query optimization on any relationship mappings. The optimization requires fewer database calls to read a set of objects from the database. Query optimization can be configured on a descriptor's mappings to affect all queries for that class. This can result in a significant system performance gain without changing any application code. Queries can also be optimized on a per-query basis. For more information, see the *TopLink: Foundation Library Guide*.

TopLink provides two query optimization features on mappings: joining and batch reading.

- Joining can be used only on one-to-one mappings. Joining joins the two related classes tables to read all of the data in a single query. This feature should be used only if it is known that the target object is always required with the source object, or indirection is not used.
- Batch reading can be used in most of the relational mappings, including direct collection mappings, one-to-one mappings, aggregate collection mappings, one-to-many mappings, and many-to-many mappings. This feature should be used only if it is known that the related objects are always required with the source object.

Example 6–5 Query Optimization Examples

The following code example illustrates using joining for query optimization.

```
// Queries for Employee are configured to always join Address
OneToOneMapping addressMapping = new OneToOneMapping();
addressMapping.setReferenceClass(Address.class);
addressMapping.setAttributeName("address");
addressMapping.useJoining();
addressMapping.privateOwnedRelationship();
```

The following code example illustrates using batch for query optimization.

```
// Queries on Employee are configured to always batch read Address
OneToManyMapping phoneNumbersMapping = new OneToManyMapping();
phoneNumbersMapping.setReferenceClass("
PhoneNumber.class")
phoneNumbersMapping.setAttributeName("phones");
phoneNumbersMapping.useBatchReading();
phoneNumbersMapping.privateOwnedRelationship();
```

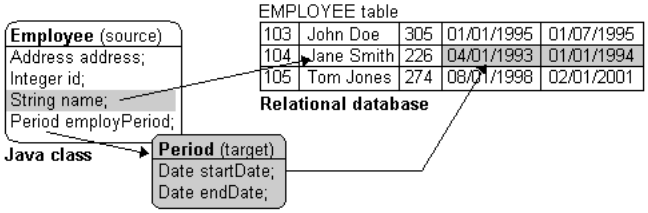
Working with Aggregate Object Mappings

Two objects are related by aggregation if there is a strict one-to-one relationship between the objects and all the attributes of the second object can be retrieved from the same table(s) as the owning object. This means that if the source (parent) object exists, then the target (child or owned) object must also exist, as illustrated in Figure 6–6.

Aggregate objects are privately owned and should not be shared or referenced by other objects.

Note: When using an aggregate descriptor in an inheritance, *all* the descriptors in the inheritance tree must be aggregates. Aggregate and Class descriptors cannot exist in the same inheritance tree.

Figure 6–6 Aggregate Object Mapping



103	John Doe	305	01/01/1995	01/07/1995
104	Jane Smith	226	04/01/1993	01/01/1994
105	Tom Jones	274	08/01/1998	02/01/2001

To implement an aggregate object mapping:

- The descriptor of the target class must declare itself to be an aggregate object. Because all of its information comes from its parent’s table(s), the target descriptor does not have a specific table associated with it. You must, however, choose one or more candidate table(s) from which you can use fields in mapping the target. In the example above, you could choose the `EMPLOYEE` table so that the `START_DATE` and `END_DATE` fields are available during mapping.

- The descriptor of the source class must add an aggregate object mapping that specifies the target class. In the example above, the `Employee` class has an attribute called `employPeriod` that would be mapped as an aggregate object mapping with `Period` as the reference class. The source class must ensure that its table has fields that correspond to the field names registered with the target class.

- If a source object has a null target reference, TopLink writes NULLs to the aggregate database fields. When the source is read from the database, it can handle this null target in one of two ways:

  - Create an instance of the object with all of its attributes equal to null.

  - Put a null reference in the source object without instantiating a target. (This is the default method of handling null targets.)

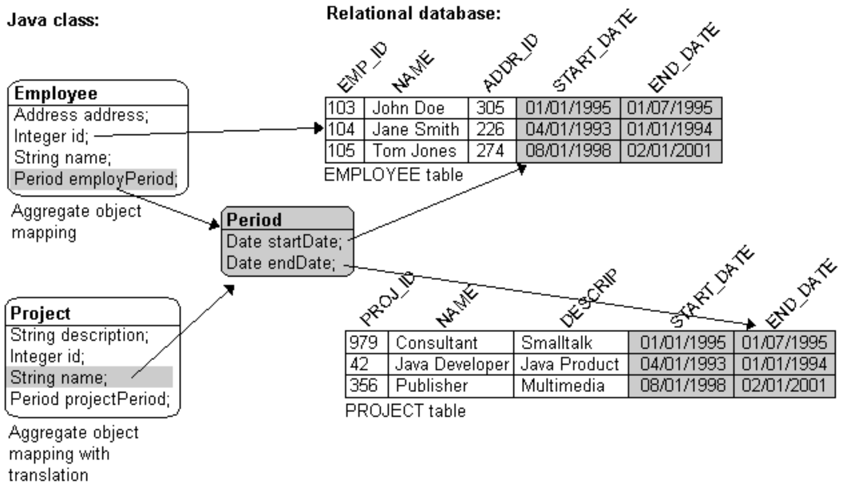
Target objects can also have multiple sources, hence the need to choose a candidate table during its mapping. This allows different source types to store the same target information within their tables. Each source class must have table fields that correspond to the field names registered with the target class. If one of the source tables has different field names than the names registered with the target class, the source class must translate the field names.

In Figure 6–7:

- The `Period` class has a direct-to-field mapping between `startDate` and `START_DATE`.

- The `Employee` class can use the `Period` class as a normal aggregate to write to its `START_DATE` column.
- The `PROJECT` table does not have a field called `START_DATE`, so the `Project` descriptor must provide a field translation on its aggregate object mapping from `START_DATE` to `S_DATE`. (If the `PROJECT` table had a `START_DATE` column, this field translation would be unnecessary.)

Figure 6–7 Aggregation with Multiple Source Classes



EMP_ID	NAME	ADDR_ID	START_DATE	END_DATE
103	John Doe	305	01/01/1995	01/07/1995
104	Jane Smith	226	04/01/1993	01/01/1994
105	Tom Jones	274	08/01/1998	02/01/2001

PROJ_ID	NAME	DESCRIP	START_DATE	END_DATE
979	Consultant	Smalltalk	01/01/1995	01/07/1995
42	Java Developer	Java Product	04/01/1993	01/01/1994
356	Publisher	Multimedia	08/01/1998	02/01/2001

Aggregate target classes that are not shared among multiple source classes can have any type of mapping, including other aggregate object mappings.

Aggregate target classes that are shared with multiple source classes cannot have one-to-many or many-to-many mappings.

Other classes cannot reference the aggregate target with one-to-one, one-to-many, or many-to-many mappings. If the aggregate target has a one-to-many relationship with another class, the other class must provide a one-to-one relationship back to the aggregate's parent class instead of the aggregate child. This is because the source class contains the table and primary key information of the aggregate.

Aggregate descriptors can make use of inheritance. The subclasses must also be declared as aggregate and be contained in the source's table. For more information on inheritance, see "Working with Inheritance" on page 4-30.

Creating a Target Descriptor

Use this procedure to create a target descriptor to use with an aggregate mapping. You must configure the target before specifying field translations in the parent descriptor.

To create the target descriptor:

1. In the **Project Tree** pane, right-click on the target descriptor and select **Aggregate** from the pop-up menu. The descriptor’s icon in the **Project Tree** pane changes to an Aggregate Descriptor .

   You can also create the target descriptor by selecting **Selected > Aggregate** from the menu or by clicking the **Aggregate Descriptor** button .

2. Map the attributes, specifying all but field information.

   - For a one-to-one mapping, pick a reference between a table in the target descriptor and a table in a descriptor that will have a mapping to this aggregate target. If this aggregate target will be mapped to by multiple source descriptors, pick a reference whose foreign key field(s) will be in the tables of one of the source descriptors.
   - For a one-to-many mapping or a many-to-many mapping, pick a reference whose foreign key field(s) will be in the referenced descriptor’s tables and whose primary key field will be in the source descriptor’s tables.

3. Continue with "Creating an Aggregate Object Mapping" on page 6-17 to create the aggregate mapping.

Creating an Aggregate Object Mapping

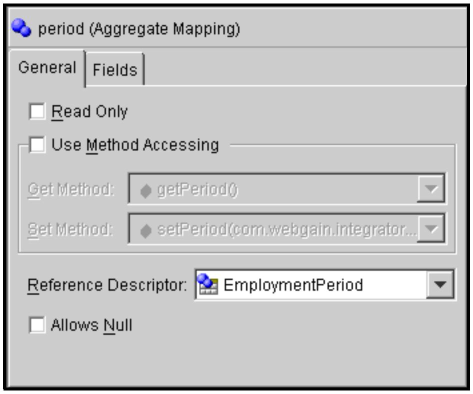
Use this procedure to create an aggregate object mapping. You must also create a target descriptor to use with the aggregate mapping.

To create an aggregate object mapping:

1. In the **Project Tree** pane, select the mapping to be mapped and click the **Aggregate Mapping** button from the mapping toolbar.

   The Aggregate mapping window appears in the **Properties** pane.

Figure 6–8 Aggregate Mapping General Tab



□ Read Only

□ Use Method Accessing

Get Method: getPeriod()

Set Method: setPeriod(com.webgain.integrator...

Reference Descriptor: EmploymentPeriod

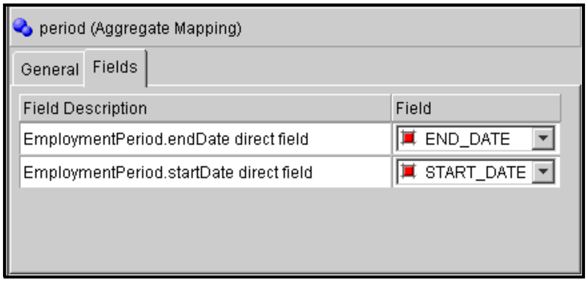
□ Allows Null

2. Use the **Reference Descriptor** drop-down list on the **General** tab to select a reference descriptor.

> **Note:** You may select only aggregate descriptors. See "Creating a Target Descriptor" on page 17.

3. You can also specify:
   - Read-only attributes – See "Specifying Read-only Settings" on page 4-63.
   - Access methods – See "Specifying Direct Access and Method Access" on page 4-62.
   - Null values – See "Defaulting Null Values" on page 4-64.
4. Click on the **Fields** tab to specify field information for the target descriptor's mapping.

Figure 6–9 Aggregate Mapping Fields Tab



period (Aggregate Mapping)

Field Description	Field
EmploymentPeriod.endDate direct field	END_DATE
EmploymentPeriod.startDate direct field	START_DATE

5. Use this table to enter data in each field:

Field	Description
Field Description	Available fields from the reference descriptor. These fields are for display only and cannot be changed on this screen.
Field	Use the drop-down list to select a field to use for the mapping for each field description.

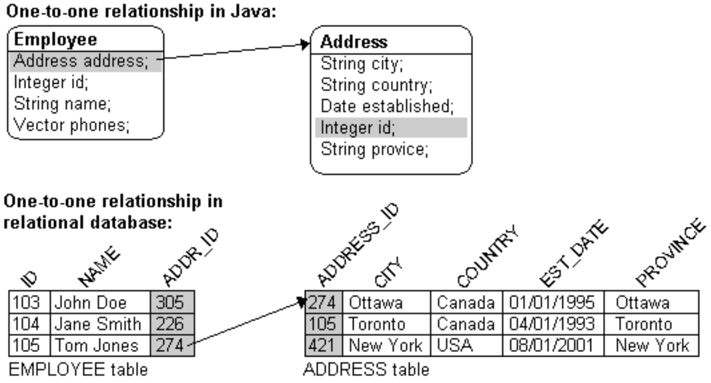
Working with One-to-one Mappings

One-to-one mappings represent simple pointer references between two Java objects. In Java, a single pointer stored in an attribute represents the mapping between the source and target objects. Relational database tables implement these mappings using foreign keys.

Figure 6–10 illustrates a one-to-one relationship from the `address` attribute of an `Employee` object to an `Address` object. To store this relationship in the database, create a one-to-one mapping between the `address` attribute and the `Address` class. This mapping stores the `id` of the `Address` instance in the `EMPLOYEE` table when the `Employee` instance is written. It also links the `Employee` instance to the `Address` instance when the `Employee` is read from the database. Since an `Address` does not have any references to the `Employee`, it does not have to provide a mapping to `Employee`.

For one-to-one mappings, the source table normally contains a foreign key reference to a record in the target table. In Figure 6–10, the ADDR_ID field of the EMPLOYEE table is a foreign key.

Figure 6–10 One-to-one Mappings



ID	NAME	ADDR_ID
103	John Doe	305
104	Jane Smith	226
105	Tom Jones	274

ADDRESS_ID	CITY	COUNTRY	EST_DATE	PROVINCE
274	Ottawa	Canada	01/01/1995	Ottawa
105	Toronto	Canada	04/01/1993	Toronto
421	New York	USA	08/01/2001	New York

You can also implement a one-to-one mapping where the target table contains a foreign key reference to the source table. In the example, the database design would change such that the `ADDRESS` row would contain the `EMP_ID` to identify the `Employee` to which it belonged. In this case, the target must also have a relationship mapping to the source.

The update, insert and delete operations for privately owned one-to-one relationships, which are normally done for the target before the source, are performed in the opposite order when the target owns the foreign key. Target foreign keys normally occur in bidirectional one-to-one mappings, as one side has a foreign key and the other shares the same foreign key in the other's table.

Target foreign keys can also occur when large cascaded composite primary keys exist (that is, one object's primary key is composed of the primary key of many other objects). In this case it is possible to have a one-to-one mapping that contains both foreign keys and target foreign keys.

In a foreign key, TopLink automatically updates the foreign key value in the object's row. In a target foreign key, it does not. In TopLink, the **Target Foreign Key** checkbox includes a checkmark when a target foreign key relationship is defined.

When mapping a relationship, it is important to understand these differences between a foreign key and a target foreign key, to ensure that the relationship is defined correctly.

In a bi-directional relationship where the two classes in the relationship reference each other, only one of the mappings should have a foreign key. The other mapping should have a target foreign key. If one of the mappings in a bi-directional relationship is a one-to-many mapping, see "Working with Variable One-to-one Mappings" on page 6-23 for details.

Creating One-to-one Mappings

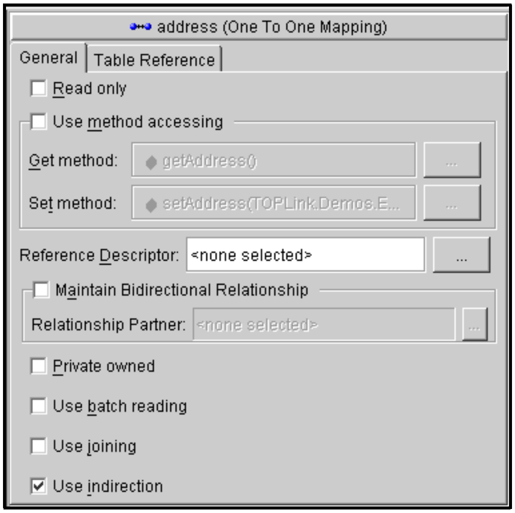
Use this procedure to create a one-to-one mapping.

To create a one-to-one mapping

1. In the **Project Tree** pane, select the mapping to be mapped and click the **One-to-One Mapping** button from the mapping toolbar.

   The One-to-one mapping window appears in the **Properties** pane.

Figure 6–11 One-to-one Mapping General Properties



address (One To One Mapping)

General | Table Reference

☐ Read only

☐ Use method accessing

Get method: getAddress() ...

Set method: setAddress(TOPLink.Demos.E... ...

Reference Descriptor: <none selected> ...

☐ Maintain Bidirectional Relationship

Relationship Partner: <none selected> ...

☐ Private owned

☐ Use batch reading

☐ Use joining

☑ Use indirection

2. Enter the required information on the **General** tab (see "Working with Common Mapping Properties" on page 4-61).

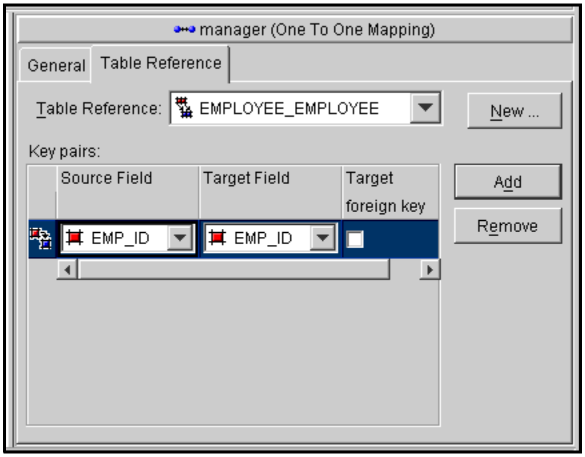
3. You can also specify:

   - Bidirectional relationships – See "Maintaining Bidirectional Relationships" on page 4-64
   - Read-only attributes – See "Specifying Read-only Settings" on page 4-63
   - Access methods – See "Specifying Direct Access and Method Access" on page 4-62

- Null values – See "Defaulting Null Values" on page 4-64

4. Click on the **Table Reference** tab to choose the reference.

Figure 6–12 One-to-one Mapping Table Reference Properties



manager (One To One Mapping)

Table Reference: EMPLOYEE_EMPLOYEE

Key pairs:

Source Field	Target Field	Target foreign key
EMP_ID	EMP_ID	□

5. Use this table to enter data in each field:

Field	Description
Table Reference	Use the drop-down list to select a table reference for the mapping. Click on **New** to create a new table
Key Pairs	
Source Field	Use the drop-down list to select a field from the source table.
Target Field	Use the drop-down list to select a field from the target table.
Target Foreign Key	Specify if the relationship is a target foreign key.

Specifying Advanced Features Available by Amending the Descriptor

One-to-one target objects mapped as **Privately Owned** are, by default, verified before deletion or update outside of a unit of work.

Verification is a check for the previous value of the target and is accomplished through joining the source and target tables. Inside a unit of work, verification is accomplished by obtaining the previous value from the back-up clone, so this setting is not used because a database read is not required. You may wish to disable verification outside of a unit of work for performance reasons and can do so by sending the `setShouldVerifyDelete()` message to the mapping in an amendment method written for the descriptor as follows:

```
public static void addToDescriptor(Descriptor descriptor){
//Find the one-to-one mapping for the address attribute
OneToOneMapping addressMapping=(OneToOneMapping)
descriptor.getMappingForAttributeName("address");
addressMapping.setShouldVerifyDelete(false);
}
```

Working with Variable One-to-one Mappings

Variable class relationships are similar to polymorphic relationships except that in this case the target classes are not related via inheritance (and thus not good candidates for an abstract table) but via an interface.

To define variable class relationships in TopLink Mapping Workbench, use the variable one-to-one mapping selection but choose the interface as the reference class. This makes the mapping a variable one-to-one. When defining mappings in Java code, use the `VariableOneToOneMapping` class.

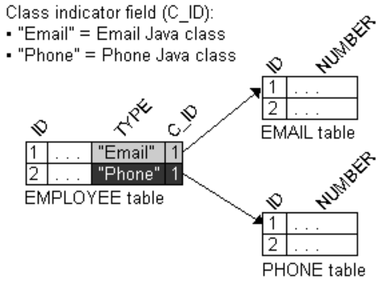
TopLink only supports variable relationship in one to one mappings. It handles this relationship in two ways:

- Through the class indicator field
- Through unique primary key values among target classes implementing the interface

Specifying Class Indicator

Through the class indicator field, a source table has an indicator column that specifies the target table, as illustrated in the following illustration. The EMPLOYEE table has a TYPE column that indicates the target for the row (either PHONE or EMAIL).

Figure 6–13 Class indicator Field



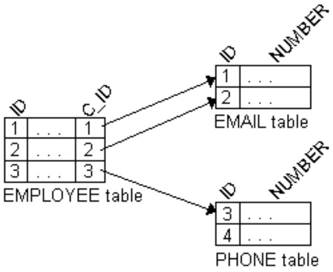
The principles of defining such a variable class relationship are similar to defining a normal one-to-one relationship, except:

- The reference class is a Java *interface*, not a Java *class*. However, the method to set the interface is identical.
- A type indicator field must be specified.
- The class indicator values are specified on the mapping so that mapping can determine the class of object to create.
- The foreign key names and the respective abstract query keys from the target interface descriptor must be specified.

Specifying Unique Primary Key

As shown in Figure 6–14, the value of the foreign key in the source table mapped to the primary key of the target table is unique. No primary key values among the target tables are the same, so primary key values are not unique just in the table, but also among the tables.

Figure 6–14 Unique primary key



Because there is no indicator stored in the source table, TopLink can not determine to which target table the foreign key value is mapped. Therefore, TopLink reads through all the target tables until it finds an entry in one of the target tables. This is an inefficient way of setting up a relation model, because reading is very expensive. The class indicator is much more efficient and it reduces the number of reads done on the tables to get the data. In the class indicator method, TopLink knows exactly which target table to look into for the data.

The principles of defining such a variable class relationship is similar to defining class indicator variable one-to-one relationships, except:

- A type indicator field is not specified.
- The class indicator values are not specified.

The type indicator field and its values are not needed, as TopLink will go through all the target tables until data is finally found.

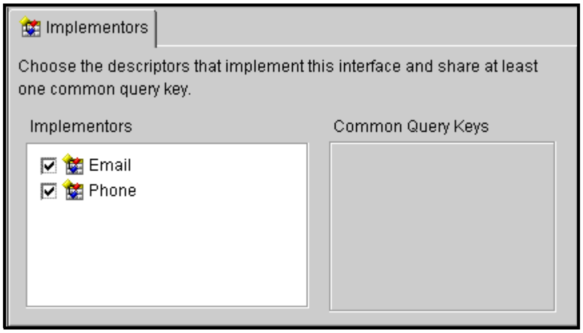
Creating Variable One-to-one Mappings

Use this procedure to create a variable one-to-one mapping. You must configure the target descriptor before defining the mapping.

To create a variable one-to-one mapping:

1. In the **Project Tree** pane, choose the interface descriptor that will be referenced.
2. On the **Implementors** tab, choose all descriptors that implement this interface and share a common query key. You may need to create query keys for some or all of these descriptors.

Figure 6–15 Implementors Tab



Implementors

Choose the descriptors that implement this interface and share at least one common query key.

Implementors

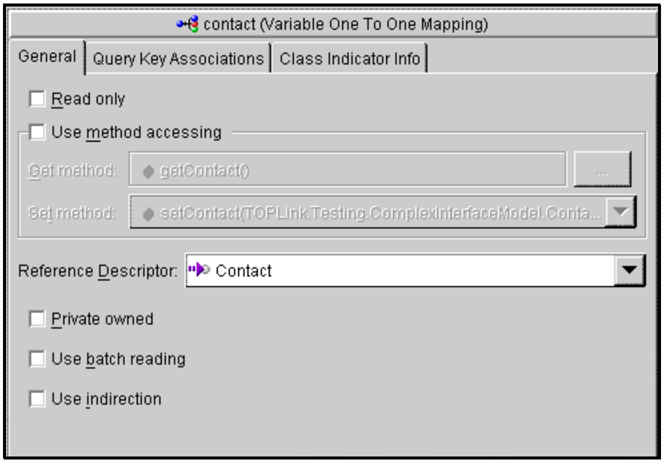
☑ Email
☑ Phone

Common Query Keys

3. In the **Project Tree** pane, choose the attribute to be mapped as a variable one-to-one mapping and click the **Variable One-to-One Mapping** button on the mapping toolbar.

4. Choose the **General** tab.

Figure 6–16 Variable one-to-one Mapping General Properties



contact (Variable One To One Mapping)

General | Query Key Associations | Class Indicator Info

□ Read only

□ Use method accessing

Get method: getContact()

Set method: setContact(TOPLink.Testing.ComplexInterfaceModel.Conta...

Reference Descriptor: Contact

□ Private owned

□ Use batch reading

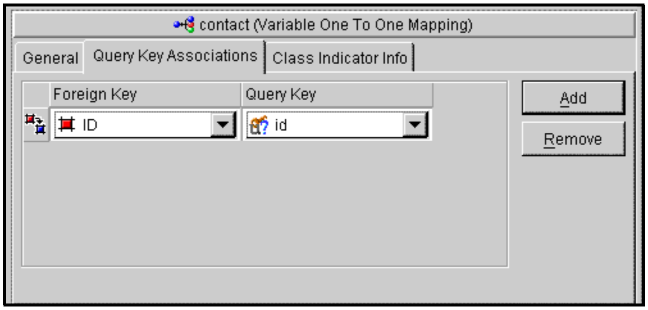
□ Use indirection

5. Use the **Reference Descriptor** drop-down list to choose a reference descriptor. The Mapping Workbench will only display interface descriptors.

6. Enter any other required information on the **General** tab (see "Working with Common Mapping Properties" on page 4-61).

7. Choose the **Query Key Associations** tab.

Figure 6–17 Variable one-to-one Mapping Query Key Associations Properties

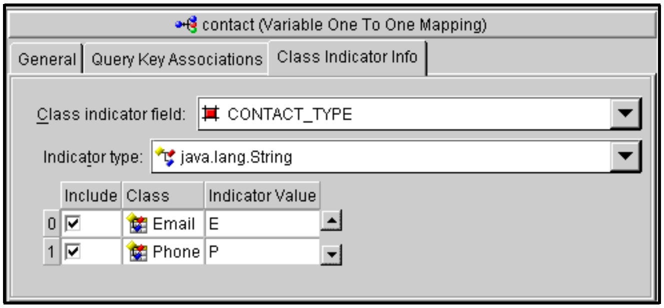


	Foreign Key	Query Key
	ID	id

8. Specify fields in the source descriptor’s tables to use for common query keys.

9. Choose the **Class Indicator Info** tab.

Figure 6–18 Variable One-to-one Mapping Class Indicator Info Tab



Class indicator field: CONTACT_TYPE

Indicator type: java.lang.String

	Include	Class	Indicator Value
0	☑	Email	E
1	☑	Phone	P

10. Use this table to enter data in each field.

Field	Description
Class Indicator Field	Use the drop-down list to select a field to use as a class indicator. To use unique primary keys (no class indicator values), select **<none selected>**.
Indicator Type	Use the drop-down list to select the Java type for the **Class Indicator Field**.
Class information:	
Include	Specify to use this class for the mapping.
Class	Name of the class. This field is for display only.

Field	Description
Indicator Value	Value used by this class.

Note: If the class does not appear in the **Class Information** table, you must add the class in the interface descriptor. See "Implementing an Interface" on page 4-39 for more information.

Working with Direct Collection Mappings

Direct collection mappings store collections of Java objects that are not TopLink-enabled. The object type stored in the direct collection is typically a Java type such as `String`.

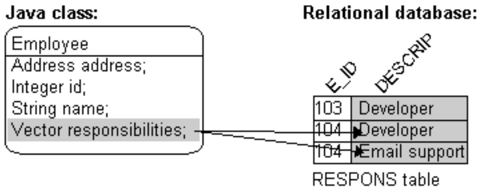
It is also possible to use direct collection mappings to map a collection of non-`String` objects. For example, it is possible to have an attribute that contains a collection of `Integer` or `Date` instances. The instances stored in the collection can be any type supported by the database and has a corresponding wrapper class in Java.

Support for primitive data types such as `int` is not provided since Java vectors only hold objects.

Example 6–6 Direct Collection Example

Figure 6–19 illustrates how a direct collection is stored in a separate table with two fields. The first field is the reference key field, which contains a reference to the primary key of the instance owning the collection. The second field contains an object in the collection and is called the direct field. There is one record in the table for each object in the collection.

Figure 6–19 Direct Collection Mappings



Note: The "responsibilities" attribute is a `Vector`. When using JDK 1.2, it is possible to use a `Collection` interface (or any class that implements the `Collection` interface) for declaring the collection attribute. See "Working with a Container Policy" on page 6-4 for details.

Maps are not supported for direct collection as there is no key value.

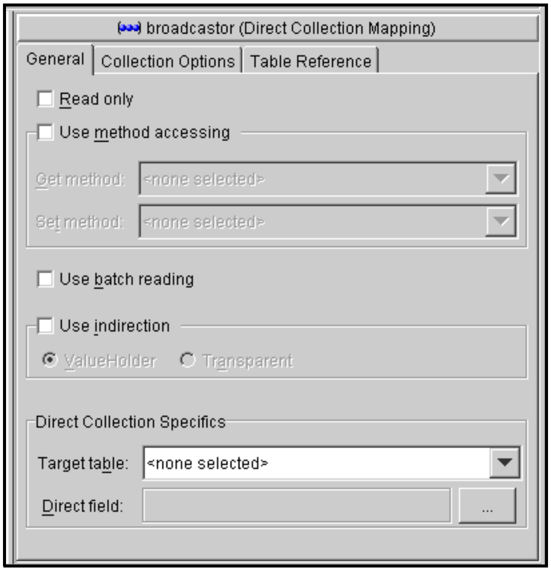
Creating Direct Collection Mappings

Use this procedure to create a direct collection mapping.

To create a direct collection mapping:

1. Select the attribute to be mapped from the **Project Tree** pane.
2. Click the **Direct Collection Mapping** button on the mapping toolbar.

Figure 6–20 Direct Collection Mapping General Properties



broadcastor (Direct Collection Mapping)

General | Collection Options | Table Reference

□ Read only

□ Use method accessing

Get method: <none selected>

Set method: <none selected>

□ Use batch reading

□ Use indirection

● ValueHolder ○ Transparent

Direct Collection Specifics

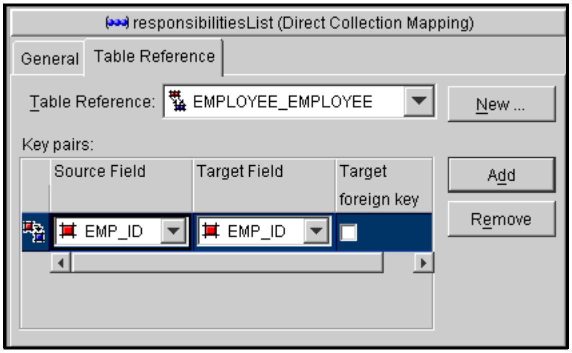
Target table: <none selected>

Direct field:

3. Use the **Target Table** and **Direct Field** drop-down lists to specify the appropriate information.

4. Enter any other required information on the **General** tab (see "Working with Common Mapping Properties" on page 4-61).
5. Choose the **Collection Options** tab to specify collection information for this mapping. See "Specifying Collection Properties" on page 4-64 for more information.
6. Choose the **Table References** tab to specify foreign key information for this mapping. See "Creating table references" on page 3-10 for more information.
7. Click on the **Table Reference** tab.

Figure 6–21 Direct Collection Mapping Table Reference Properties



responsibilitiesList (Direct Collection Mapping)

Table Reference: EMPLOYEE_EMPLOYEE

Key pairs:

Source Field	Target Field	Target foreign key
EMP_ID	EMP_ID	□

8. Choose the appropriate reference that relates the target table to the tables associated with the source descriptor.

Working with Aggregate Collection Mappings

Aggregate collection mappings are used to represent the aggregate relationship between a single-source object and a collection of target objects. Unlike the TopLink one-to-many mappings, in which there should be a one-to-one back reference mapping from the target objects to the source object, there is no back reference required for the aggregate collection mappings because the foreign key relationship is resolved by the aggregation.

> **Caution:** Aggregate collections are not directly supported in the Mapping Workbench. You must use an amendment method (see "Amending Descriptors After Loading" on page 4-18) or manually edit the project source to add the mapping.

To implement an aggregate collection mapping:

- The descriptor of the target class must declare itself to be an aggregate collection object. Unlike the aggregate object mapping, in which the target descriptor does not have a specific table to associate with, there must be a target table for the target object.
- The descriptor of the source class must add an aggregate collection mapping that specifies the target class.

Aggregate collection descriptors can use inheritance. The subclasses must also be declared as aggregate collection. The subclasses can have their own mapped tables, or share the table with their parent class. For more information on inheritance, see "Working with Inheritance" on page 4-30.

In a Java `Vector`, the owner references its parts. In a relational database, the parts reference their owners. Relational databases use this implementation to make querying more efficient.

> **Note:** For information on using collection classes other than `Vector` with aggregate collection mappings, see *Oracle9iAS TopLink: Foundation Library Guide*.

Working with One-to-many Mappings

One-to-many mappings are used to represent the relationship between a single source object and a collection of target objects. They are a good example of something that is simple to implement in Java using a Vector (or other collection types) of target objects, but difficult to implement using relational databases.

In a Java Vector, the owner references its parts. In a relational database, the parts reference their owner. Relational databases use this implementation to make querying more efficient.

> **Note:** See "Working with a Container Policy" on page 6-4 for information on using collection classes other than Vector with one-to-many mappings.

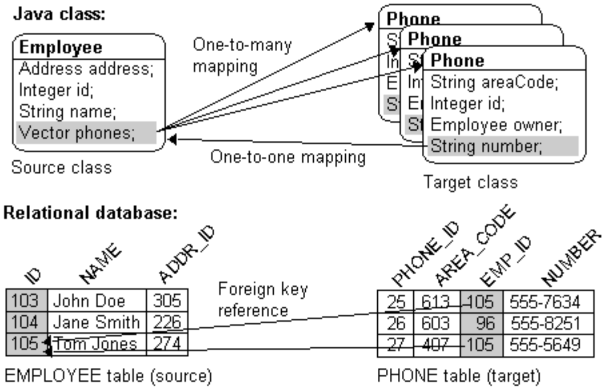
The purpose of creating this one-to-one mapping in the target is so that the foreign key information can be written when the target object is saved. Alternatives to the one-to-one mapping back reference include:

- Use a direct-to-field mapping to map the foreign key and maintain its value in the application. Here the object model does not require a back reference, but the data model still requires a foreign key in the target table.
- Use a many-to-many mapping to implement a logical one-to-many. This has the advantage of not requiring a back reference in the object model and not requiring a foreign key in the data model. In this model the many-to-many relation table stores the collection. It is possible to put a constraint on the join table to enforce that the relation is a logical one-to-many relationship.

Example 6–7 One-to-many Mapping Example

One-to-many mappings must put the foreign key in the target table, rather than the source table. The target class should also implement a one-to-one mapping back to the source object, as illustrated in the following figure.

Figure 6–22 One-to-many Relationships



ID	NAME	ADDR_ID
103	John Doe	305
104	Jane Smith	226
105	Tom Jones	274

PHONE_ID	AREA_CODE	EMP_ID	NUMBER
25	613	105	555-7634
26	603	96	555-8251
27	407	105	555-5649

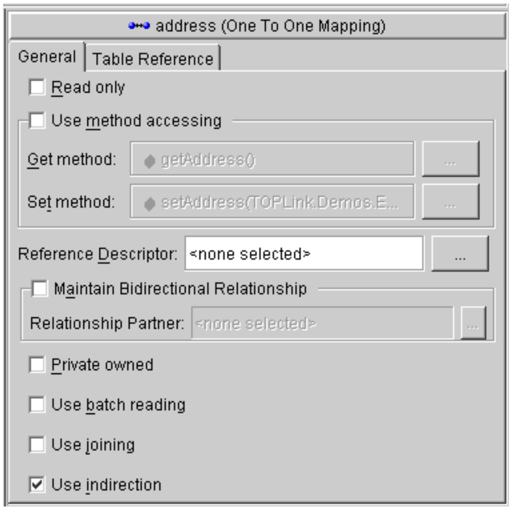
Creating One-to-many Mappings

Use this procedure to create a one-to-many mapping in the Mapping Workbench.

To create a one-to-many mapping:

1. Select the attribute to be mapped from the **Project Tree** pane.
2. Click the **One-to-Many Mapping** button on the mapping toolbar.

Figure 6–23 One-to-many mapping General Properties



☐ Read only

☐ Use method accessing

Get method: getAddress()

Set method: setAddress(TOPLink.Demos.E...

Reference Descriptor: <none selected>

☐ Maintain Bidirectional Relationship

Relationship Partner: <none selected>

☐ Private owned

☐ Use batch reading

☐ Use joining

☑ Use indirection

3. Use the **Reference Descriptor** drop-down list to select the reference for this descriptor.
4. You can also specify:
   - Bidirectional relationships – See "Maintaining Bidirectional Relationships" on page 4-64
   - Read-only attributes – See "Specifying Read-only Settings" on page 4-63
   - Access methods – See "Specifying Direct Access and Method Access" on page 4-62
   - Null values – See "Defaulting Null Values" on page 4-64
5. Choose the **Collection Options** tab to specify collection information for this mapping. See “Specifying Collection Properties” on page 64 for more information.
6. Choose the **Table References** tab to specify foreign key information for this mapping. See “Creating table references” on page 10 for more information.

Working with Many-to-many Mappings

Many-to-many mappings represent the relationships between a collection of source objects and a collection of target objects. They require the creation of an intermediate table for managing the associations between the source and target records. Figure 6–24 and Figure 6–24 illustrate a many-to-many mapping in Java and in relational database tables.

Many-to-many mappings are implemented using a relation table. This table contains columns for the primary keys of the source and target tables. Composite primary keys require a column for each field of the composite key. The intermediate table must be created in the database before using the many-to-many mapping.

The target class does not have to implement any behavior for the many-to-many mappings. If the target class also creates a many-to-many mapping back to its source, it can use the same relation table, but one of the mappings must be set to read-only. If both mappings write to the table they can cause collisions.

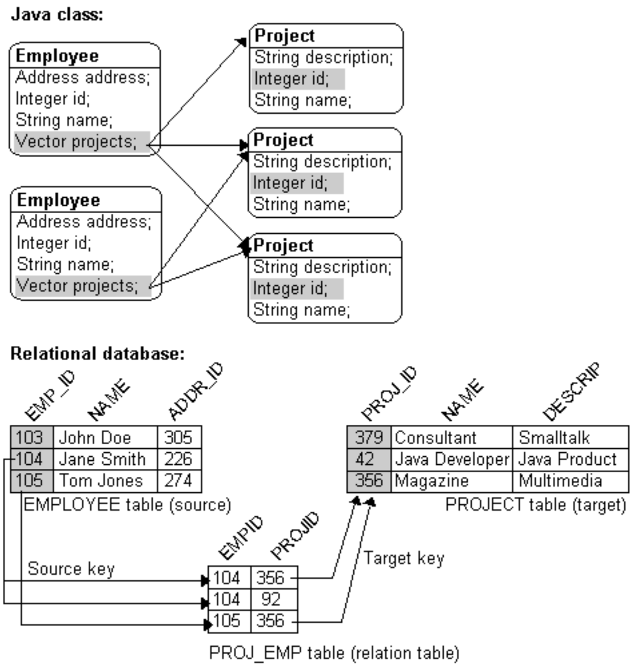
> **Note:** See "Working with a Container Policy" on page 6-4 for information on using collection classes other than Vector with one-to-many mappings.

Indirection is enabled by default in a many-to-many mapping, which requires that the attribute have the `ValueHolderInterface` type or transparent collections.

Example 6–8 Many-to-many Example

The following figures illustrate a many-to-many relationship in both Java and a relational database.

Figure 6–24 Many-to-many Relationships



EMP_ID	NAME	ADDR_ID
103	John Doe	305
104	Jane Smith	226
105	Tom Jones	274

EMPLOYEE table (source)

PROJ_ID	NAME	DESCRIP
379	Consultant	Smalltalk
42	Java Developer	Java Product
356	Magazine	Multimedia

PROJECT table (target)

EMPID	PROJID
104	356
104	92
105	356

PROJ_EMP table (relation table)

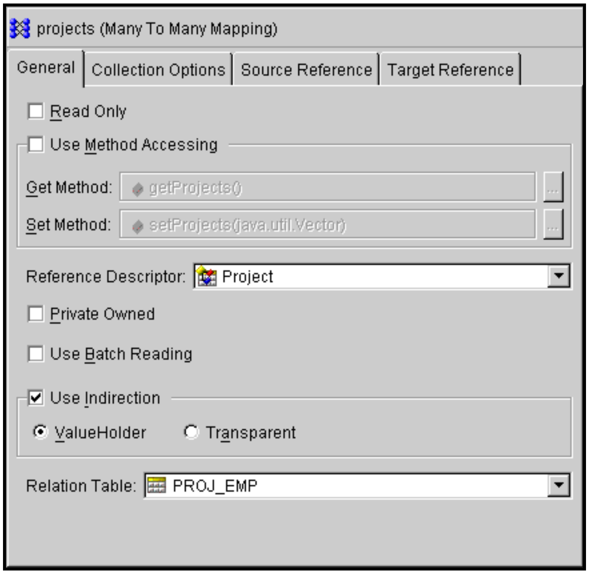
Creating many-to-many Mappings

Use this procedure to create a many-to-many mapping.

To create a many-to-many mapping:

1. In the **Project Tree** pane, select the attribute to be mapped.
2. Click the **Many-to-Many Mapping** button in the mapping toolbar.

Figure 6–25 Many-to-many Mapping General Properties



☐ Read Only

☐ Use Method Accessing

Get Method: getProjects()

Set Method: setProjects(java.util.Vector)

Reference Descriptor: Project

☐ Private Owned

☐ Use Batch Reading

☑ Use Indirection

● ValueHolder ○ Transparent

Relation Table: PROJ_EMP

3. Use the **Reference Descriptor** drop-down list to choose the reference descriptor for this mapping.
4. Use the **Relation Table** drop-down list to select the relation table.
5. Modify any other properties, as needed. See "Working with Common Mapping Properties" on page 4-61 for more information.
6. Click on the **Collection Options** tab to specify the source descriptor relates to the relation table. See "Specifying Collection Properties" on page 4-64 for more information.
7. Click on the **Source Reference** tab to specify how the source descriptor relates to the relation table.

Figure 6–26 Many-to-many Mapping Source Reference Properties

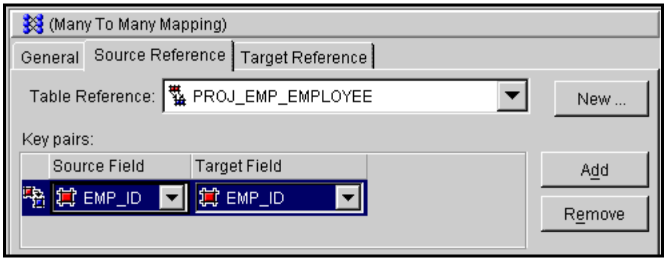


Table Reference: PROJ_EMP_EMPLOYEE

Key pairs:

Source Field	Target Field
EMP_ID	EMP_ID

8. Use the **Table Reference** drop-down list to choose a reference whose foreign key is in the relation table and that points to a table associated to the source descriptor. See "Creating table references" on page 3-10 for more information.
9. Click on the **Target Reference** tab to specify how the reference descriptor relates to the relation table.
10. Choose a reference whose foreign key is in the relation table and that points to a table associated to the reference descriptor. See "Creating table references" on page 3-10 for more information.

Specifying Advanced Features by Amending the Descriptor

TopLink can populate a collection in ascending or descending order upon your specification. To do this, specify and write an amendment method, sending the `addAscendingOrdering()` or `addDescendingOrdering()` to the many-to-many mapping. Both messages expect a string as a parameter, which indicates what attribute from the target object is used for the ordering. This string can be an attribute name or query key from the target's descriptor. Query keys are automatically created for and with the same name as all attributes mapped as direct-to-field, type conversion, object type, and serialized object mappings.

Example 6–9 Descriptor Amendment Example

The following code example illustrates returning an Employee's projects in ascending order according to their descriptions

```
public static void addToDescriptor(Descriptor descriptor)
{
//Find the Many-to-Many mapping for the projects attribute
ManyToManyMapping projectsMapping=(ManyToManyMapping)
descriptor.getMappingForAttributeName("projects");
projectsMapping.addAscendingOrdering("description");
}
```

Working with Custom Relationship Mappings

Just as a descriptor's query manager generates the default SQL code that is used for database interaction, relationship mappings also generate query information.

Like the queries used by a descriptor's query manager, queries associated with relationship mappings can be customized using SQL strings or query objects. Refer to "Specifying Queries" on page 4-12 for more information on customizing queries and the syntax that TopLink supports.

To customize the way a relationship mapping generates SQL:

- selection — All relationship mappings can use the `setSelectionCriteria()`, `setSelectionSQLString()`, and `setCustomSelectionQuery()` methods of the mapping to customize the selection criteria.
- insert — Many-to-many and direct collection mappings can use the `setInsertSQLString()` or `setCustomInsertQuery()` methods of the mapping to customize the insertion criteria.
- delete all — Many-to-many, direct collection, and one-to-many mappings can use the `setDeleteAllSQLString()` and `setCustomDeleteAllQuery()` methods of the mapping to customize the deletion criteria.
- delete —Many-to-many mappings can use the `setDeleteSQLString()` and `setCustomDeleteQuery()` methods of the mapping to customize the deletion criteria.

A query object that specifies the search criteria must be passed to each of these methods. Because search criteria for these operations usually depend on variables at runtime, the query object must usually be created from a parameterized expression, SQL string, or stored procedure call.

See *TopLink: Foundation Library Guide* for more information on defining parameterized queries and stored procedure calls.

Creating Custom Mapping Queries in Java Code

The following example illustrates selection customization with a parameterized expression using `setSelectionCriteria()` and deletion customization using `setDeleteAllSQLString()`. Because the descriptor is passed as the parameter to this amendment method, which has been specified to be called after the descriptor is loaded in the project, we must locate each mapping for which we wish to define a custom query.

Example 6–10 Custom Mapping Example

The following code illustrates adding a custom query to two different mappings in the `Employee` descriptor.

```
// Amendment method in Employee class
public static void addToDescriptor(Descriptor descriptor)
{

//Find the one-to-one mapping for the address attribute
OneToOneMapping addressMapping=(OneToOneMapping)
descriptor.getMappingForAttributeName("homeaddress");

//Create a parameterized Expression and register it as the default selection
criterion for the mapping.
ExpressionBuilder builder = new ExpressionBuilder();
addressMapping.setSelectionCriteria(builder.getField("ADDRESS.ADDRESS_
ID").equal(builder.getParameter("EMP.ADDRESS_
ID")).and(builder.getField("ADDRESS.TYPE").equal("home")));

// Get the direct collection mapping for responsibilitiesList.
DirectCollectionMapping directCollection=(DirectCollectionMapping)
descriptor.getMappingForAttributeName("responsibilitiesList");
directCollection.setDeleteAllSQLString("DELETE FROM RESPONS WHERE EMP_ID = #EMP_
ID");
}
```

7

Understanding Object Relational Mappings

Relational mappings define how persistent objects reference other persistent objects. Oracle 9*i*AS TopLink supports the following object relational mapping types:

- *Array mappings* are similar to direct collection mappings but map to object-relational array data-types (the Array type in JDBC 2.0 and the VARRAY type in Oracle 8*i*). Use array mappings to map a collection of primitive data. See "Working with Array Mappings" on page 7-2 for more information.
- *Object array mappings* are similar to array mappings but map to object-relational array data types. See "Working with Object Array Mappings" on page 7-4 for more information.
- *Structure mappings* are similar to aggregate object mappings but map to object-relational aggregate structures (the Struct type in JDBC 2.0 and the OBJECT TYPE in Oracle 8*i*). See "Working with Structure Mappings" on page 7-5 for more information.
- *Reference mappings* are similar to one-to-one mappings but map to object-relational references (the Ref type in JDBC 2.0 and the REF type in Oracle 8*i*). See "Working with Reference Mappings" on page 7-7 for more information.
- *Nested table mappings* are similar to many-to-many mappings but map to object-relational nested tables (the NESTED TABLE type in Oracle 8*i*). See "Working with Nested Table Mappings" on page 7-9 for more information.

These mappings allow for an object model to be persisted into an object-relational data-model. Currently the Mapping Workbench does not support object-relational mappings – they must be defined in code or through amendment methods. See "Working with Object-relational Descriptors" on page 4-59 for more information.

Working with Object Relational Mappings

Object relational mappings allow for an object model to be persisted into an object-relational data-model. The Mapping Workbench does not directly support these mappings – they must be defined in code through amendment methods.

TopLink supports the following object relational mappings:

- Array
- Object array
- Structure
- Reference
- Nested table

Working with Array Mappings

In an object-relational data-model, structures can contain *arrays* (collections of other data types). These arrays can contain primitive data types or collections of other structures. TopLink stores the arrays with their parent structure in the same table.

All elements in the array must be the same data type. The number of elements in an array controls the size of the array. An Oracle database allows arrays of variable sizes (called Varrays).

Oracle8*i* provides two collection types:

- Varray – Used to represent a collection of primitive data or aggregate structures.
- Nested table – Similar to varrays except they store information in a separate table from the parent structure’s table

TopLink supports arrays of primitive data through the `ArrayMapping`. This is similar to `DirectCollectionMapping` – it represents a collection of primitives in Java. However, the `ArrayMapping` does not require an additional table to store the values in the collection.

TopLink supports arrays of aggregate structures through the `ObjectArrayMaping`.

TopLink supports nested tables through the `NestedTableMapping`.

Implementing Array Mappings in Java

Array mappings are instances of the ArrayMapping class. You must associate this mapping to an attribute in the parent class. TopLink requires the following elements for an array mapping:

- Attribute being mapped – Set by sending the setAttributeName( ) message.
- Field being mapped – Set by sending the setFieldName( ) message.
- Name of the array – Set by sending the setStructureName( ) message.

Table 7–1 summarizes all array mapping properties:

Example 7–1 Array Mapping Example

The following code example illustrates creating an array mapping for the Employee source class and registering it with the descriptor

```
// Create a new mapping and register it with the source descriptor.
ArrayMapping arrayMapping = new ArrayMapping();
arrayMapping.setAttributeName("responsibilities");
arrayMapping.setStructureName("Responsibilities_t");
arrayMapping.setFieldName("RESPONSIBILITIES");
descriptor.addMapping(arrayMapping);
```

Reference

The following table summarizes all array mapping properties. In the Method Names column, arguments are **bold**, methods are not.

Table 7–1 Properties for ArrayMapping methods

Property	Default	Method Names
Attribute to be mapped *	not applicable	setAttributeName**(String name)**
Set parent class *	not applicable	setReferenceClass**(Class referenceClass)**
User-defined data type *	not applicable	setStructureName**(String Structurename)**
Field to be mapped *	not applicable	setFieldName**(String fieldName)**

** Required property*

Table 7–1 Properties for ArrayMapping methods(Cont.)

Property	Default	Method Names

Method access	direct access	`setGetMethodName`**`(String name)`**
		`setSetMethodName`**`(String name)`**

Read only	read / write	`readWrite`**`()`**
		`readOnly`**`()`**
		`setIsReadOnly`**`(boolean readOnly)`**

** Required property*

Working with Object Array Mappings

In an object-relational data-model, object arrays allow for an array of object types or structures to be embedded into a single column in a database table or an object table.

TopLink supports object array mappings to define a collection-aggregated relationship in which the target objects share the same row as the source object.

Implementing Object Array Mappings in Java

Object array mappings are instances of the `ObjectArrayMapping` class. You must associate this mapping to an attribute in the parent class. TopLink requires the following elements for an array mapping:

- Attribute being mapped – Set by sending the `setAttributeName( )` message.
- Field being mapped – Set by sending the `setFieldName( )` message.
- Name of the array – Set by sending the `setStructureName( )` message.

Use the optional `setGetMethodName( )` and `setSetMethodName( )` messages to access the attribute through user-defined methods rather than directly. See "Specifying Direct Access and Method Access" on page 4-62 for more information.

Table 7–2 summarizes all object array mapping properties.

Example 7–2 Object Array Mapping Example

The following code example illustrates creating an object array mapping for the `Insurance` source class and registering it with the descriptor.

```
// Create a new mapping and register it with the source descriptor.
ObjectArrayMapping phonesMapping = new ObjectArrayMapping();
```

```
phonesMapping.setAttributeName("phones");
phonesMapping.setGetMethodName("getPhones");
phonesMapping.setSetMethodName("setPhones");
phonesMapping.setStructureName("PHONELIST_TYPE");
phonesMapping.setReferenceClass(Phone.class);
phonesMapping.setFieldName("PHONES");
descriptor.addMapping(phonesMapping);
```

Reference

The following table summarizes all object array mapping properties. In the Method Names column, arguments are **bold**, methods are not.

Table 7–2 Properties for ObjectArrayMapping Methods

Property	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName`**`(String name)`**
Set parent class *	not applicable	`setReferenceClass`**`(Class referenceClass)`**
User-defined data type *	not applicable	`setStructureName`**`(String structureName)`**
Field to be mapped *	not applicable	`setFieldName`**`(String fieldName)`**

Method access	direct access	`setGetMethodName`**`(String name)`**
		`setSetMethodName`**`(String name)`**

Read only	read / write	`readWrite`**`()`**
		`readOnly`**`()`**
		`setIsReadOnly`**`(boolean readOnly)`**

** Required property*

Working with Structure Mappings

In an object-relational data-model, structures are user defined data-types or object-types. This is similar to a Java class – it denies attributes or fields in which each attribute is either:

- a primitive data type
- another structure

- reference to another structure

TopLink maps each structure to a Java class defined in your object model and defines a descriptor for each class. A StructureMapping maps nested structures, similar to an AggregateObjectMapping. However, the structure mapping supports null values and shared aggregates without requiring additional settings (because of the object-relational support of the database).

Implementing Structure Mappings in Java

Structure mappings are instances of the StructureMapping class. You must associate this mapping to an attribute in each of the parent classes. TopLink requires the following elements for an array mapping:

- Attribute being mapped – Set by sending the setAttributeName( ) message.
- Field being mapped – Set by sending the setFieldName( ) message.
- Target (child) class – Set by sending the setReferenceClass( ) message.

Use the optional setGetMethodName( ) and setSetMethodName( ) messages to access the attribute through user-defined methods rather than directly. See "Specifying Direct Access and Method Access" on page 4-62 for more information.

You must make the following changes to the target (child) class descriptor:

- Send the descriptorIsAggregate() message to indicate it is not a root level.
- Remove table or primary key information.

Table 7–3 summarizes all structure mapping properties:

Example 7–3 Structure Mapping Examples

The following code example illustrates creating a structure mapping for the Employee source class and registering it with the descriptor

```
// Create a new mapping and register it with the source descriptor.
StructureMapping structureMapping = new StructureMapping();
structureMapping.setAttributeName("address");
structureMapping.setReferenceClass(Address.class);
structureMapping.setFieldName("address");
descriptor.addMapping(structureMapping);
```

The following code example illustrates creating the descriptor of the Address aggregate target class. The aggregate target descriptor does not need a mapping to its parent, or any table or primary key information.

```
// Create a descriptor for the aggregate class. The table name and primary key
are not specified in the aggregate descriptor.
ObjectRelationalDescriptor descriptor = new ObjectRelationalDescriptor ();
descriptor.setJavaClass(Address.class);
descriptor.setStructureName("ADDRESS_T");
descriptor.descriptorIsAggregate();
// Define the field ordering
descriptor.addFieldOrdering("STREET");
descriptor.addFieldOrdering("CITY");
...
// Define the attribute mappings or relationship mappings.
...
```

Reference

The following table summarizes all structure mapping properties. In the Method Names column, arguments are **bold**, methods are not.

Table 7–3 Properties for StructureMapping Methods

Property	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName`**`(String name)`**
Set parent class *	not applicable	`setReferenceClass`**`(Class aClass)`**
Field to be mapped *	not applicable	`setFieldName`**`(String fieldName)`**

Method access	direct access	`setGetMethodName`**`(String name)`**
		`setSetMethodName`**`(String name)`**

Read only	read / write	`readWrite`**`()`**
		`readOnly`**`()`**
		`setIsReadOnly`**`(boolean readOnly)`**

** Required property*

Working with Reference Mappings

In an object-relational data-model, structures reference each other through *refs* – not through foreign keys (as in a traditional data-model). Refs are based on the target structure's `ObjectID`.

TopLink supports refs through the ReferenceMapping. They represent an object reference in Java, similar to a OneToOneMapping. However, the reference mapping does not require foreign key information.

Implementing Reference Mappings in Java

Reference mappings are instances of the ReferenceMapping class. You must associate this mapping to an attribute in the source class. TopLink requires the following elements for a reference mapping:

- Attribute being mapped – Set by sending the setAttributeName( ) message.
- Field being mapped – Set by sending the setFieldName( ) message.
- Target class – Set by sending the setReferenceClass( ) message.

Use the optional setGetMethodName( ) and setSetMethodName( ) messages to access the attribute through user-defined methods rather than directly. See "Specifying Direct Access and Method Access" on page 4-62 for more information.

Table 7–4 summarizes all reference mapping properties.

Example 7–4 Reference Mapping Example

The following code example illustrates creating a reference mapping for the Employee source class and registering it with the descriptor.

```
// Create a new mapping and register it with the source descriptor.
ReferenceMapping refrenceMapping = new ReferenceMapping();
referenceMapping.setAttributeName(“manager”);
referenceMapping.setReferenceClass(Employee.class);
referenceMapping.setFieldName("MANAGER");
descriptor.addMapping(refrenceMapping);
```

Reference

The following table summarizes all reference mapping properties. In the Method Names column, arguments are **bold**, methods are not.

Table 7–4 Properties for ReferenceMapping Methods

Property	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Set parent class *	not applicable	`setReferenceClass(Class aClass)`
Field to be mapped *	not applicable	`setFieldName(String fieldName)`

Method access	direct access	`setGetMethodName(String name)`
		`setSetMethodName(String name)`

Indirection	use indirection	`useBasicIndirection()`
		`dontUseIndirection()`

Privately owned relationship	independent	`independentRelationship()`
		`privateOwnedRelationship()`
		`setIsPrivateOwned(boolean isPrivateOwned)`

Read only	read / write	`readWrite()`
		`readOnly()`
		`setIsReadOnly(boolean readOnly)`

** Required property*

Working with Nested Table Mappings

Nested table types model an unordered set of elements. These elements may be built-in or user-defined types. You can view a nested table as a single-column table or, if the nested table is an object type, as a muticolumn table (with a column for each attribute of the object type).

Typically, nested tables represent a one-to-many or many-to-many relationship of references to another independent structure. They support querying and joining better than Varrays that are inlined to the parent table.

TopLink supports nested table through the `NestedTableMapping`. They represent a collection of object references in Java, similar to a `OneToManyMapping` or `ManyToManyMapping`. However, the nested table mapping does not require foreign key information (like a one-to-many mapping) or the relational table (like a many-to-many mapping).

Implementing Nested Table Mappings in Java

Nested table mappings are instances of the `NestedTableMapping` class. This mapping is associated to an attribute in the parent class. The following elements are required for a nested table mapping to be viable:

- The attribute being mapped, which is set by sending the `setAttributeName()` message
- The field being mapped, which is set by sending the `setFieldName()` message
- The name of the array structure, which is set by sending the `setStructureName()` message

Use the optional `setGetMethodName()` and `setSetMethodName()` messages to allow TopLink to access the attribute through user-defined methods rather than directly. See "Specifying Direct Access and Method Access" on page 4-62 for more information.

Table 7–5 summarizes all nested table mapping properties.

Example 7–5 Nested Table Example

The following code example illustrates creating a nested table mapping for the Insurance source class and registering it with the descriptor.

```
// Create a new mapping and register it with the source descriptor.
NestedTableMapping policiesMapping = new NestedTableMapping();
policiesMapping.setAttributeName("policies");
policiesMapping.setGetMethodName("getPolicies");
policiesMapping.setSetMethodName("setPolicies");
policiesMapping.setReferenceClass(Policy.class);
policiesMapping.dontUseIndirection();
policiesMapping.setStructureName("POLICIES_TYPE");
policiesMapping.setFieldName("POLICIES");
policiesMapping.privateOwnedRelationship();
policiesMapping.setSelectionSQLString("select p.* from policyHolders ph,
table(ph.policies) t, policies p where ph.ssn=#SSN and ref(p) = value(t)");
descriptor.addMapping(policiesMapping);
```

Reference

The following table summarizes all nested table mapping properties. In the Method Names column, arguments are **bold**, methods are not.

Table 7–5 Properties for NestedTableMapping Methods

Property	Default	Method Names
Attribute to be mapped *	not applicable	`setAttributeName(String name)`
Set parent class *	not applicable	`setReferenceClass(Class referenceClass)`
User-defined data type *	not applicable	`setStructureName(String structureName)`
Field to be mapped *	not applicable	`setFieldName(String fieldName)`

Method access	direct access	`setGetMethodName(String name)`
		`setSetMethodName(String name)`

Indirection	use indirection	`useIndirection()`
		`dontUseIndirection()`
		`setUsesIndirection(boolean usesIndirection)`

Privately owned relationship	independent	`independentRelationship()`
		`privateOwnedRelationship()`
		`setIsPrivateOwned(Boolean isPrivateOwned)`

Read only	read / write	`readWrite()`
		`readOnly()`
		`setIsReadOnly(boolean readOnly)`

** Required property*

A

Object Model Requirements

Oracle 9*i*AS TopLink requires that classes must meet certain minimum requirements before they can become *persistent*. TopLink also provides alternatives to most requirements. TopLink uses a non-intrusive approach using a meta-data architecture that allows for almost no object model intrusions.

This section summarizes TopLink's object model requirements. Unlike other products, TopLink *does not* require any of the following:

- Persistent superclass or implementation of persistent interfaces
- Stored, delete or load methods required in the object model
- Special persistence methods
- Generating source code into or wrapping the object model

Persistent Class Requirements

The attribute requirements vary, depending on your Java version. When using Java 2, you can use direct access on private or protected attributes. Refer to Chapter 4, "Understanding Descriptors" for more information on direct and method access.

When using *non-transparent* indirection, the attributes must be of type `ValueHolderInterface` rather than the original attribute type. The value holder does not instantiate a referenced object until it is needed.

In Java 2, TopLink provides *transparent* indirection for `Collection` and `List` attribute types for any collection mappings. Using transparent indirection does not require the usage of `ValueHolderInterface`, or any other object model requirements.

Refer to Chapter 6, "Understanding Relationship Mappings" for more information on indirection and transparent indirection.

Constructor Requirements

By default, TopLink uses and requires default (zero argument) constructors to create objects from the database. It is also possible to instruct TopLink to use a different constructor, static method, or factory. Refer to "Working with Instantiation Policy" on page 4-44 for more information.

Remote Session Requirements

If the TopLink Remote Session is used, all persistent classes to be used remotely must implement the `Serializable` interface.

Index