Oracle® Database

JPublisher User's Guide

10*g* Release 1 (10.2)

B14188-01

August 2006

ORACLE®

Oracle Database JPublisher User's Guide, 10*g* Release 1 (10.2)

B14188-01

Primary Author: Venkatasubramaniam Iyer, Brian Wright

Contributing Authors: Janice Nygard, Thomas Pfaeffle, Ekkehard Rohwedder, P. Alan Thiesen

Contributor: Deepa Aswani, Prabha Krishna, Ellen Siegal, Quan Wang

Contents

2 Using JPublisher

3 Data Type and Java-to-Java Type Mappings

4 Generated Classes and Interfaces

5 Additional Features and Considerations

6 Command-Line Options and Input Files

List of Tables

Preface

This preface introduces you to the *Oracle Database JPublisher User's Guide*, discussing the intended audience, structure, and conventions of this document. A list of related Oracle documents is also provided.

The JPublisher utility is for Java programmers who want classes in their applications to correspond to SQL or PL/SQL entities or server-side Java classes. In Oracle Database 10*g*, JPublisher also provides features supporting Web services call-ins to the database and call-outs from the database.

This preface covers the following topics:

Intended Audience

The *Oracle Database JPublisher User's Guide* is intended for Java Database Connectivity (JDBC) and Java2 Platform, Enterprise Edition (J2EE) programmers who want to accomplish any of the following for database applications:

- Create Java classes to map to SQL user-defined types, including object types, `VARRAY` types, and nested table types
- Create Java classes to map to `OPAQUE` types
- Create Java classes to map to PL/SQL packages
- Create client-side Java stubs to call server-side Java classes
- Publish SQL queries or data manipulation language (DML) statements as methods in Java classes
- Create Java and PL/SQL wrappers for Web services client proxy classes
- Publish server-side SQL, PL/SQL, or Java entities as Web services

To use this document, you need knowledge of Java, Oracle Database, SQL, PL/SQL, and JDBC.

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. Accessibility standards will continue to evolve over time, and Oracle 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 more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers 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 does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Structure

This document contains:

Chapter 1, "Introduction to JPublisher"

Introduces the JPublisher utility by way of examples, lists new features in this release, and provides an overview of JPublisher operations.

Chapter 2, "Using JPublisher"

Describes how you can use JPublisher for publishing SQL and PL/SQL objects, Oracle Stream Advanced Queue (AQ), server-side Java classes, and Web services.

Chapter 3, "Data Type and Java-to-Java Type Mappings"

Provides details of JPublisher data type mappings and the styles mechanism for Java-to-Java type mappings.

Chapter 4, "Generated Classes and Interfaces"

Discusses details and concepts of the classes, interfaces, and subclasses generated by JPublisher, including how output parameters (PL/SQL `IN OUT` or `OUT` parameters) are treated, how overloaded methods are translated, and how the generated classes and interfaces are used.

Chapter 5, "Additional Features and Considerations"

Covers additional JPublisher features and considerations: a summary of support for Web services, filtering of JPublisher output, and migration and backward compatibility.

Chapter 6, "Command-Line Options and Input Files"

Provides details of the JPublisher command-line syntax, command-line options and their usage, and input file format.

Appendix A, "Generated Code Examples"

Contains code examples that are too lengthy to fit conveniently with corresponding material earlier in the manual. This includes examples of Java-to-Java type transformations to support Web services, and Java and PL/SQL wrappers to support Web services.

Appendix B, "Troubleshooting"

Covers the troubleshooting tips for JPublisher

Related Documents

For more information, see the following Oracle resources.

From the Oracle Java Platform group, for Oracle Database releases:

- *Oracle Database Java Developer's Guide*

  This book introduces the basic concepts of Java in Oracle Database and provides general information about server-side configuration and functionality. It contains information that pertains to the Oracle Database Java environment in general, rather than to a particular product, such as JDBC.

  The book also discusses Java stored procedures, which are programs that run directly in Oracle Database. With stored procedures (functions, procedures, and triggers), Java developers can implement business logic at the server level, which improves application performance, scalability, and security.

- *Oracle Database JDBC Developer's Guide and Reference*

  This book covers programming syntax and features of the Oracle implementation of the JDBC standard. This includes an overview of the Oracle JDBC drivers, the details of the Oracle implementation of JDBC 1.22, 2.0, and 3.0 features, and a discussion of Oracle JDBC type extensions and performance extensions.

From the Oracle Java Platform group, for Oracle Application Server releases:

- *Oracle Application Server Containers for J2EE Developer's Guide*
- *Oracle Application Server Containers for J2EE Services Guide*
- *Oracle Application Server Containers for J2EE Security Guide*
- *Oracle Application Server Containers for J2EE Servlet Developer's Guide*
- *Oracle Application Server Containers for J2EE Support for JavaServer Pages Developer's Guide*
- *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*
- *Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide*

From the Oracle Server Technologies group:

- *Oracle XML DB Developer's Guide*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database XML Java API Reference*
- *Oracle Database Application Developer's Guide - Fundamentals*
- *Oracle Database Application Developer's Guide - Large Objects*
- *Oracle Database Application Developer's Guide - Object-Relational Features*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database PL/SQL User's Guide and Reference*
- *Oracle Database SQL Reference*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database Advanced Security Administrator's Guide*
- *Oracle Database Globalization Support Guide*
- *Oracle Database Reference*

Note: Oracle error message documentation is available in HTML only. Only if you have access to the Oracle Documentation CD, you can browse the error messages by range. After you find the specific range, use the Find in Page feature of your browser to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle online documentation.

From the Oracle Application Server group:

- *Oracle Application Server 10g Administrator's Guide*
- *Oracle HTTP Server Administrator's Guide*
- *Oracle Application Server 10g Performance Guide*
- *Oracle Application Server 10g Globalization Guide*
- *Oracle Application Server Web Cache Administrator's Guide*
- *Oracle Application Server Web Services Developer's Guide*
- *Oracle Application Server 10g Upgrading to 10g (9.0.4)*

From the Oracle JDeveloper group:

- JDeveloper online help
- JDeveloper documentation on the Oracle Technology Network:

  `http://otn.oracle.com/products/jdev/content.html`

Printed documentation is available for sale in the Oracle Store at

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

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/membership/`

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

`http://otn.oracle.com/documentation/`

For additional information, see:

`http://jcp.org/aboutJava/communityprocess/final/jsr101/index.html`

The preceding link provides access to the *Java API for XML-based RPC, JAX-RPC 1.0* specification, with information about JAX-RPC and holders.

`http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/native2ascii.html`

For Java Development Kit (JDK) users, the preceding link contains `native2ascii` documentation, including information about character encoding that is supported by Java environments.

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

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

Convention	Meaning	Example
Bold	Bold typeface indicates terms defined in the text.	A **class** is a blueprint that defines the variables and the methods common to all objects of a certain kind.
Italics	Italic typeface indicates book titles and emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates system elements. Such elements include parameters, privileges, data types (including user-defined types), RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, user names, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executables, file names, directory names, and some user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as Java packages and classes, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to start SQL*Plus. The password is specified in the `orapwd` file. Back up the data files and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. 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 *old_release*`.SQL` where *old_release* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate Java, SQL, PL/SQL, SQL*Plus, or 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. Do not enter the braces.	`{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;`
Other notation	You must enter symbols other than brackets, braces, 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*

Convention	Meaning	Example

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case-sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

1

Introduction to JPublisher

This chapter provides an overview of the JPublisher utility, a summary of JPublisher operations, and a sample translation. It covers the following topics:

Overview of JPublisher

JPublisher is a utility written in Java. It generates Java classes to represent database entities, such as SQL objects and PL/SQL packages, in a Java client program. It also provides support for publishing from SQL, PL/SQL, and server-side Java to Web services and enabling the invocation of external Web services from inside the database.

JPublisher can create classes to represent the following database entities types:

- User-defined SQL objects
- Object references
- User-defined SQL collections
- PL/SQL packages
- Server-side Java classes
- SQL queries and data manipulation language (DML) statements

JPublisher enables you to specify and customize the mapping of these entities to Java classes in a strongly typed paradigm.

> **Note:** The term, **strongly typed**, indicates that a particular Java type is associated with a particular user-defined SQL type, such as an object type. For example, a `Person` class is associated with a corresponding `PERSON` SQL type. In addition, there is a corresponding Java type for each attribute of the SQL object type.

The utility generates the accessor methods, `get`*`XXX`*`()` and `set`*`XXX`*`()`, for each attribute of an object type. If the object type has stored procedures, then JPublisher can

generate wrapper methods to invoke the stored procedures. In this scenario, a wrapper method is a Java method that invokes a stored procedure, which runs in the Oracle Database.

JPublisher can also generate classes for PL/SQL packages. These classes have wrapper methods to call the stored procedures in a PL/SQL package.

Instead of directly using JPublisher-generated classes, you can:

- Extend the generated classes. This process is straightforward. JPublisher generates initial versions of the subclasses, to which you can add any desired functionality.
- Write your own Java classes. This approach is flexible, but time-consuming and error-prone.
- Use generic, weakly typed classes of the `oracle.sql` package to represent object, object reference, and collection types. If these classes meet your requirements, then you do not need JPublisher. Use this approach if you need to generically process any SQL object, collection, reference, or `OPAQUE` type.

In addition, JPublisher simplifies access to PL/SQL types from Java. You can use predefined or user-defined mappings between PL/SQL and SQL types, as well as use PL/SQL conversion functions between such types. With these mappings in place, JPublisher can automatically generate all the required Java and PL/SQL code. It also enables you to publish server-side Java classes to client-side Java classes, allowing your application to make direct calls to Java classes in the database.

Several features enable the exposure of Java classes, which are generated from publishing SQL, PL/SQL, or server-side Java entities, as Web services.

> **See Also:** "Summary of JPublisher Support for Web Services" on page 5-1

JPublisher Initial Considerations

The following sections provide an overview of new JPublisher features and requirements and how JPublisher uses SQLJ in its code generation:

New Features in Oracle Database 10*g* JPublisher

New features in Oracle Database 10*g* release 1 (10.1) JPublisher include support for:

- Generation of Java interfaces
- Style files for Java-to-Java type mappings
- `REF CURSOR` returning and result set mapping
- Filtering what JPublisher publishes
- Publishing server-side Java classes
- Publishing SQL queries or SQL DML statements
- Web services call-outs from the database

The following JPublisher features have been introduced in Oracle Database 10*g* release 2 (10.2):

- Support for JDBC types in server-side Java call-ins
- Support for publishing Oracle Streams Advanced Queue (AQ) as Web services
- Support for complex types in Web services call-outs
- Generation of SQLJ run time free code

The key new features in Oracle Database 10*g* JPublisher can be categorized as follows:

- New Features for Web Services
- Awareness of Java Environment
- Implicit SQLJ Translation

New Features for Web Services

The new features of JPublisher enable the exposure of generated Java classes as Web services for invocation from outside the database. There are two stages to implementing this:

1. Publishing from SQL or PL/SQL to Java using JPublisher
2. Publishing from Java to Web services using the Oracle Web services assembler tool

You can also load and wrap client proxy classes, so that external Web services can be called from Java or PL/SQL inside the database.

JPublisher also provides support for publishing Oracle Streams AQ. This feature exposes a queue or a topic as a Java program, which can be published as a Web service by the Web service assembler.

> **See Also:** "Summary of JPublisher Support for Web Services" on page 5-1 and Chapter 2, "Using JPublisher"

Awareness of Java Environment

UNIX releases of JPublisher, prior to Oracle Database 10*g*, ignore the `CLASSPATH` environment variable. Instead, they use a classpath provided through the JPublisher command line that includes the required JPublisher and Java Database Connectivity (JDBC) classes. In Oracle Database 10*g*, the value of the `CLASSPATH` environment variable is appended to the classpath that is provided through the command line.

In Oracle Database 10*g*, JPublisher picks up the `CLASSPATH` environment variable on all platforms. This ensures successful processing of classes when JPublisher loads user-provided types, such as for Web services call-outs, and conversion of Java types during publishing. The term **Web services call-outs** refers to calling Web services from inside the database by loading the Web services client proxies into the database and generating Java and PL/SQL wrappers for these client proxies. In contrast, the term **Web services call-ins** refers to the functionality of having SQL, PL/SQL, and server-side Java classes in Oracle Database that are accessible to Web services clients.

Implicit SQLJ Translation

In Oracle Database 10*g*, JPublisher generates SQLJ code and implicitly translates the generated SQLJ code into Java code. JPublisher, by default, compiles the generated Java code into Java class files. By default, the intermediate SQLJ code is not visible as output from JPublisher.

Note: You can configure JPublisher to generate visible SQLJ code using the backward compatibility options.

In Oracle Database 10*g*, JPublisher continues to provide options for translating and compiling SQLJ code.

See Also: "Option to Access SQLJ Functionality" on page 6-45

In addition, JPublisher provides options for run time-free generation of SQLJ code and SQLJ to JDBC migration.

See Also: "SQLJ Migration Options" on page 6-49 and `http://www.oracle.com/technology/tech/java/sqlj_jdbc/pdf/oracle_sqlj_roadmap.pdf`

JPublisher Usage of the SQLJ Implementation

This section covers the following topics:

- Overview of SQLJ Usage
- Overview of SQLJ Concepts
- Backward-Compatibility Modes Affecting SQLJ Source Files

Overview of SQLJ Usage

The Oracle SQLJ translator and run-time libraries are supplied with the JPublisher product. The JPublisher utility uses the Oracle SQLJ implementation by generating SQLJ code as an intermediate step in most circumstances, such as the creation of wrapper methods. The wrapper methods are created either for classes representing PL/SQL packages or for classes representing SQL object types that define PL/SQL stored procedures. In these cases, JPublisher uses the Oracle SQLJ translator during compilation and the Oracle SQLJ run time during program execution.

In Oracle Database 10*g*, the usage of SQLJ by JPublisher is transparent by default. SQLJ source files that JPublisher generates are automatically translated and deleted, unless you specify otherwise in the JPublisher settings. This automatic translation saves you the effort of explicitly translating the files. The resulting `.java` files, which use the SQLJ functionality, and the associated `.class` files produced by compilation, define the **SQLJ classes**. These classes use the Oracle SQLJ run-time application programming interfaces (APIs) while running. Generated classes that do not use the SQLJ run time are referred to as **non-SQLJ classes**. Non-SQLJ classes are generated when JPublisher creates classes for SQL types that do not have stored procedures or when JPublisher is specifically set to not generate wrapper methods.

In Oracle Database 10*g*, it is possible to pass options to the SQLJ translator through the JPublisher `-sqlj` option.

See Also: "Option to Access SQLJ Functionality" on page 6-45

To support its use of SQLJ, JPublisher includes `translator.jar`, which contains the JPublisher and SQLJ translator libraries, and `runtime12.jar`, which is the SQLJ run-time library for Java Development Kit (JDK) 1.2 and later.

Overview of SQLJ Concepts

A SQLJ program is a Java program containing embedded SQL statements that comply with the International Standardization Organization (ISO) standard SQLJ Language Reference syntax. SQLJ source code contains a mixture of standard Java source, SQLJ class declarations, and SQLJ executable statements with embedded SQL operations.

SQLJ was chosen because it uses simplified code for database access as compared to JDBC code. In SQLJ, a SQL statement is embedded in a single `#sql` statement, but several JDBC statements may be required for the same operation.

This section briefly defines the following key concepts of SQLJ:

- Connection contexts

  A SQLJ **connection context** object is a strongly typed database connection object. You can use each connection context class for a particular set of interrelated SQL entities. This means that all the connections you define using a particular connection context class will use tables, views, and stored procedures that have names and data types in common. In theory, the advantage in tailoring connection context classes to sets of SQL entities is in the degree of online semantics-checking that is permitted during SQLJ translation. JPublisher does not use online semantics-checking when it invokes the SQLJ translator, but you can use this feature if you choose to work with `.sqlj` files directly.

  The `sqlj.runtime.ref.DefaultContext` connection context class is used by default. The SQLJ **default context** is a default connection object and an instance of this class. The `DefaultContext` class or any custom connection context class implements the standard `sqlj.runtime.ConnectionContext` interface. You can use the JPublisher `-context` option to specify the connection context class that JPublisher must instantiate for database connections.

  See Also: "SQLJ Connection Context Classes" on page 6-20

- Iterators

  A SQLJ **iterator** is a strongly typed version of a JDBC result set and is associated with the underlying database cursor. SQLJ iterators are used for taking query results from a `SELECT` statement. The strong typing is based on the data type of each query column.

- Execution contexts

  A SQLJ **execution context** is an instance of the standard `sqlj.runtime.ExecutionContext` class and provides a context in which SQL operations are run. An execution context instance is associated either implicitly or explicitly with each SQL operation that is run through SQLJ code.

Backward-Compatibility Modes Affecting SQLJ Source Files

In Oracle8*i* Database and Oracle9*i* Database, JPublisher produces `.sqlj` source files as visible output, which you can translate by using the SQLJ command-line interface.

> **Note:** On UNIX, you can access the SQLJ command-line interface by running the `sqlj` script. In Microsoft Windows, you use `sqlj.exe`.

In Oracle Database 10*g*, JPublisher supports several backward-compatibility settings through the `-compatible` option. This option enables you to work with generated

`.sqlj` files in a similar fashion. Some of the `-compatible` option settings are as follows:

- `-compatible=sqlj`

  This forces JPublisher skip the step of translating `.sqlj` files. You must translate the `.sqlj` files explicitly. To translate the files, you can either run JPublisher using only the `-sqlj` option or you can run the SQLJ translator directly through its own command-line interface.

  See Also: "Option to Access SQLJ Functionality" on page 6-45

- `-compatible=9i`

  This sets JPublisher to the Oracle9*i* compatibility mode. In this mode, JPublisher generates `.sqlj` files with the same code as in Oracle9*i* versions. This enables you to work directly with `.sqlj` files.

- `-compatible=8i` or `-compatible=both8i`

  This sets JPublisher to the Oracle8*i* compatibility mode. JPublisher then generates `.sqlj` files with the same code as in Oracle8*i* versions. As with the Oracle9*i* compatibility mode, this mode enables you to work directly with `.sqlj` files.

Oracle8*i* and Oracle9*i* compatibility modes, particularly the former, result in significant differences in the code that JPublisher generates. If your only goal is to work directly with the `.sqlj` files, then use the `sqlj` setting.

See Also: "Backward Compatibility and Migration" on page 5-5 and "Backward Compatibility Option" on page 6-46

General Requirements for JPublisher

This section describes the basic requirements for JPublisher. It also discusses situations with less stringent requirements.

When you use the JPublisher utility, you must also have classes for the Oracle SQLJ implementation, the Oracle JDBC implementation, and a Sun Microsystems JDK, among other things.

To use all features of JPublisher, you must have the following installed and set in the appropriate environment variables, as applicable:

- Oracle Database 10*g* or Oracle9*i* Database
- JPublisher invocation script or executable

  The `jpub` script for UNIX or the `jpub.exe` program for Microsoft Windows must be in your file path. These are typically in *`ORACLE_HOME`*`/bin`, or *`ORACLE_HOME`*`/sqlj/bin` for manual downloads. With proper setup, if you type just `jpub` on the command line, you will see information about common JPublisher options and input settings.

- JPublisher and SQLJ translator classes

  These classes are in the `translator.jar` library, typically in *`ORACLE_HOME`*`/sqlj/lib`.

  Note: The translator library is also automatically loaded into the database in `translator-jserver.jar`.

- SQLJ run time classes

  The SQLJ run-time library is `runtime12.jar` for JDK 1.2 and later. It is typically located in *ORACLE_HOME*`/sqlj/lib`. It includes JPublisher client-side run-time classes, particularly `oracle.jpub.reflect.Client`, and JPublisher server-side run-time classes, particularly `oracle.jpub.reflect.Server`. These classes are used for Java call-ins to the database.

- Oracle Database 10*g* or Oracle9*i* JDBC drivers

  The Oracle JDBC library is `classes12.jar` for JDK 1.2 and later and `ojdbc14.jar` for JDK 1.4. It is typically located in *ORACLE_HOME*`/jdbc/lib`. Each JDBC library also includes the JPublisher run-time classes in the `oracle.jpub.runtime` package.

  See Also: *Oracle Database JDBC Developer's Guide and Reference*

- Web services classes

  These classes are included in the `dbwsa.jar` and `dbwsclient.jar` libraries, which are typically located in *ORACLE_HOME*`/sqlj/lib`.

  Note: These `.jar` files are not included in JPublisher distribution, but are included in the database Web services call-out utility, which can be downloaded from `http://www.oracle.com/technology/sample_code/tech/java/jsp/dbwebservices.html`.

- Additional PL/SQL packages and Java Archive (JAR) files in the database, as needed

  There are packages and JAR files that must be in the database if you use JPublisher features for Web services call-ins, Web services call-outs, support for PL/SQL types, or support for invocation of server-side Java classes. Some of these packages and files are preloaded, but some must be loaded manually.

  See Also: "Required Database Setup"

- `aurora.zip`

  When publishing a Web services client using `-proxywsdl` or publishing server-side Java classes using `-dbjava`, JPublisher may load generated Java wrapper into the database. In this case, the *ORACLE_HOME*`/lib/aurora.zip` file is required. On Microsoft Windows, add this file to `CLASSPATH`. On Unix, the `jpub` script picks up `aurora.zip` automatically. If the `aurora.zip` file is not available, then you can turn off the JPublisher loading behavior by specifying `-proxyopt=noload` on the command line.

- JDK version 1.2 or later

  For Web services call-outs or to map `SYS.XMLType` for Web services, you need JDK 1.4 or later.

  See Also: "Java Environment Options" on page 6-48

Required Database Setup

Depending on the JPublisher features that you need to use, some or all of the following PL/SQL packages and JAR files must be present in the database:

- The `SQLJUTL` package, to support PL/SQL types
- The `SQLJUTL2` package, to support invocation of server-side Java classes
- The `sqljutl.jar` file or its contents, to support Web services call-ins

  In Oracle Database 10*g*, support for Web services call-ins is preloaded in the database Java virtual machine (JVM). Therefore, the `sqljutl.jar` file is not required. In Oracle9*i* Database or Oracle8*i* Database, you must load the file manually.
- The `UTL_DBWS` package, to support Web services call-outs
- The `dbwsclient.jar` file, to support the Java API for XML-based Remote Procedure Call (JAX-RPC) or Simple Object Access Protocol (SOAP) client proxy classes for Web services call-outs from Oracle Database 10*g*.

  See Also: "Options to Facilitate Web Services Call-Outs" on page 6-41

- JAR files to support SOAP client proxy classes for Web services call-outs from Oracle9*i* or Oracle8*i* Database

  For Web services call-outs from Oracle9*i* Database or Oracle8*i* Database, there is no JAR file similar to `dbwsclient.jar`. You need to load several JAR files instead. Also note that JPublisher does not yet support JAX-RPC client proxy classes in Oracle9*i* or Oracle8*i*.

  Note: The `UTL_DBWS` package and the `dbwsclient.jar` file are associated with each other, and both support the same set of features. This is also true of the `SQLJUTL2` package and `sqljutl.jar` file. The `SQLJUTL` package and `sqljutl.jar` file, however, are not directly associated with each other. Their naming is coincidental.

This section covers the following topics:

- Verifying or Installing the SQLJUTL and SQLJUTL2 Packages
- Verifying or Loading the sqljutl.jar File
- Verifying or Installing the UTL_DBWS Package
- Verifying or Loading the dbwsclient.jar File
- Loading JAR Files For Web Services Call-Outs in Oracle9i or Oracle8i
- Setting Up Password File for Remote SYS Login

Verifying or Installing the SQLJUTL and SQLJUTL2 Packages

In Oracle Database 10*g*, the PL/SQL packages `SQLJUTL` and `SQLJUTL2` are automatically installed in the database `SYS` schema. To verify the installation, try to describe the packages as follows:

```
SQL> describe sys.sqljutl
SQL> describe sys.sqljutl2
```

If JPublisher displays a message similar to the following, then the packages are missing:

```
Warning: Cannot determine what kind of type is <schema>.<type.> You likely need
to install SYS.SQLJUTL. The database returns: ORA-06550: line 1, column 7:
PLS-00201: identifier 'SYS.SQLJUTL' must be declared
```

To install the SQLJUTL and SQLJUTL2 packages, you must install one of the following files in the SYS schema:

- *ORACLE_HOME*/sqlj/lib/sqljutl.sql for Oracle9*i* or Oracle Database 10*g*
- *ORACLE_HOME*/sqlj/lib/sqljutl8.sql for Oracle8*i*

Verifying or Loading the sqljutl.jar File

The following file or its contents must be loaded in the database for server-side Java invocation:

ORACLE_HOME/sqlj/lib/sqljutl.jar

In Oracle Database 10*g*, the contents of the file are preloaded in the JVM. In Oracle9*i* Database or Oracle8*i* Database, you must load the file manually. To see whether it is already loaded, run the following query in the SYS schema:

```
SQL> select status, object_type from all_objects where
     dbms_java.longname(object_name)='oracle/jpub/reflect/Client';
```

The following result indicates that the file has already been loaded:

```
STATUS  OBJECT_TYPE
------- ------------------
VALID   JAVA CLASS
VALID   SYNONYM
```

If it is not already loaded, then you can use the loadjava utility to load the contents of this file, as shown in the following example:

```
% loadjava -oci8 -u sys/change_on_install -r -v -f -s
           -grant public sqlj/lib/sqljutl.jar
```

> **Note:** Before loading this file, verify that the java_pool_size database parameter is set to at least 96 MB and the shared_pool_size parameter is set to at least 80 MB. If not, then update the init.ora database parameter file to set appropriate values for these two parameters and restart the database.

Verifying or Installing the UTL_DBWS Package

In Oracle Database 10*g*, the UTL_DBWS PL/SQL package is automatically installed in the database SYS schema. To verify the installation, try to describe the package as follows:

```
SQL> describe sys.utl_dbws
```

If the output indicates that the package is not yet installed, then run the following scripts under SYS:

```
ORACLE_HOME/sqlj/lib/utl_dbws_decl.sql
ORACLE_HOME/sqlj/lib/utl_dbws_body.sql
```

Verifying or Loading the dbwsclient.jar File

In Oracle Database 10*g*, the following file must be loaded into the database for Web services call-outs:

ORACLE_HOME/sqlj/lib/dbwsclient.jar

It is not preloaded, but you can verify whether it is already loaded by running the following query in the SYS schema:

```
SQL>  select status, object_type from all_objects where
      dbms_java.longname(object_name)='oracle/jpub/runtime/dbws/DbwsProxy$1';
```

The following result indicates that the file is already loaded:

```
STATUS  OBJECT_TYPE
------- ------------------
VALID   JAVA CLASS
VALID   SYNONYM
```

If it not loaded, then you can use the loadjava utility to load it as shown in the following example:

```
% loadjava -oci8 -u sys/change_on_install -r -v -f -s
           -grant public dbwsclient.jar
```

> **Note:** Before loading this file, verify that java_pool_size is set to at least 96 MB and shared_pool_size is set to at least 80 MB.

Loading JAR Files For Web Services Call-Outs in Oracle9*i* or Oracle8*i*

For Web services call-outs from an Oracle9*i* or Oracle8*i* database, use SOAP client proxy classes. For this, you must load a number of JAR files into the database. This can be accomplished with the following command:

```
% loadjava -u sys/change_on_install -r -v -s -f -grant public
           ORACLE_HOME/soap/lib/soap.jar
           ORACLE_HOME/dms/lib/dms.jar
           J2EE_HOME/lib/servlet.jar
           J2EE_HOME/lib/ejb.jar
           J2EE_HOME/lib/mail.jar
```

You can obtain these files from an Oracle Application Server installation. You would presumably run Web services in conjunction with Oracle Application Server Containers for J2EE (OC4J).

> **Note:**
>
> - The JAX-RPC client proxy classes are not yet supported in Oracle9*i* or Oracle8*i*.
> - Before loading this file, verify that java_pool_size is set to at least 96 MB and shared_pool_size is set to at least 80 MB.

Setting Up Password File for Remote SYS Login

By default, if the -user and -sysuser options are set while publishing Web services client using -proxywsdl or publishing server-side Java classes using -dbjava, then JPublisher will load the generated Java and PL/SQL wrappers into the database.

When the `-url` setting specifies a JDBC Thin driver, the loading process requires the database password file to be set up properly. You can set up the password file by performing the following steps:

1. On the command line, enter the following command:

   ```
   orapwd file=$ORACLE_HOME/dbs/orapw password=yourpass entries=5
   ```

   In the preceding command, *`yourpass`* is the password of your choice.

2. From SQL*Plus, connect to the database as `SYSDBA`, as follows:

   ```
   CONNECT / AS SYSDBA
   ```

   Change the password of `SYS` to the password set in the previous step, as follows:

   ```
   ALTER USER SYS IDENTIFIED BY yourpass
   ```

3. Edit the `init.ora` file and add the following line to it:

   ```
   REMOTE_LOGIN_PASSWORDFILE=EXCLUSIVE
   ```

   This enables remote login as `SYSDBA`.

   See Also: *Oracle Database JDBC Developer's Guide and Reference*

Alternatively, with the password file set up, you can manually load JPublisher generated PL/SQL wrapper and Java wrapper into the database. To turn off automatic loading by JPublihser, specify `-proxyopt=noload` on the command line.

Situations for Reduced Requirements

If you do not use certain features of JPublisher, then your requirements may be less stringent. Some of the situations for reduced requirements are as follows:

- If you never generate classes that implement the Oracle-specific `oracle.sql.ORAData` interface or the deprecated `oracle.sql.CustomDatum` interface, then you can use a non-Oracle JDBC driver and connect to a non-Oracle database. However, JPublisher must be able to connect to Oracle Database.

  Note: Oracle does not test or support configurations that use non-Oracle components.

- If you instruct JPublisher to *not* generate wrapper methods by setting `-methods=false`, or if your object types do not define any methods, then JPublisher will not generate wrapper methods or produce any SQLJ classes. In these circumstances, there is no SQLJ translation step and the SQLJ translator is not required.

  See Also: "Generation of Package Classes and Wrapper Methods" on page 6-32

- If you use JPublisher to generate custom object classes that implement only the deprecated `CustomDatum` interface, then you can use the Oracle8*i* release 8.1.5 database with the 8.1.5 version of the JDBC driver and JDK version 1.1 or later. But it is advisable to upgrade to the `ORAData` interface, which requires the JDBC implementation in Oracle9*i* or later.

- If you do not use JPublisher functionality for invocation of server-side Java classes, then you do not need the `sqljutl.jar` file to be loaded in the database.
- If you do not use JPublisher functionality to enable Web services call-outs, then you do not need `dbwsa.jar` or `dbwsclient.jar` to be loaded in the database.

JPublisher Limitations

You need to be aware of the following limitations when you use JPublisher:

- JPublisher support for PL/SQL `RECORD` and indexed-by table types is limited. An intermediate wrapper layer is used to map a `RECORD` or an indexed-by-table argument to a SQL type that JDBC supports. In addition, JPublisher cannot fully support the semantics of indexed-by tables. An indexed-by table is similar in structure to a Java hashtable, but information is lost when JPublisher maps this to a SQL `TABLE` type.

  See Also: "Type Mapping Support for PL/SQL RECORD and Index-By Table Types" on page 3-19

- If you use an `INPUT` file to specify type mappings, then note that some potentially disruptive error conditions do not result in error or warning messages from JPublisher. Additionally, there are reserved terms that you are not permitted to use as SQL or Java identifiers.

  See Also: "INPUT File Precautions" on page 6-56

- The `-omit_schema_names` JPublisher option has a boolean logic, but does not use the same syntax as other boolean options. You can use this option to instruct JPublisher to *not* use schema names to qualify SQL names that are referenced in wrapper classes. By default, JPublisher uses schema names to qualify SQL names. To disable the use of schema names, enter the `-omit_schema_names` option on the command line, but do *not* attempt to set `-omit_schema_names=true` or `-omit_schema_names=false`.

  See Also: "Omission of Schema Name from Name References" on page 6-33

  Note: This chapter refers to the input file specified by the `-input` option as the `INPUT` file to distinguish from any other kinds of input files.

What JPublisher Can Publish

You can use JPublisher to publish:

- SQL user-defined types
- PL/SQL packages
- Server-side Java classes
- SQL queries or DML statements
- Proxy classes and wrappers for Web services call-outs
- Oracle Streams AQ

See Also: Chapter 2, "Using JPublisher"

JPublisher Mappings and Mapping Categories

The following sections provide a basic overview of JPublisher mappings and mapping categories:

- JPublisher Mappings for User-Defined Types and PL/SQL Types
- JPublisher Mapping Categories

JPublisher Mappings for User-Defined Types and PL/SQL Types

JPublisher provides mappings from the following to Java classes:

- User-defined SQL types
- PL/SQL types

Representing User-Defined SQL Types Through JPublisher

You can use an Oracle-specific implementation, a standard implementation, or a generic implementation in representing user-defined SQL types, such as objects, collections, object references, and `OPAQUE` types, in your Java program.

Following is a summary of these three approaches:

- Use classes that implement the Oracle-specific `ORAData` interface.

  JPublisher generates classes that implement the `oracle.sql.ORAData` interface. The `ORAData` interface supports SQL objects, object references, collections, and `OPAQUE` types in a strongly typed way. That is, for each specific object, object reference, collection, or `OPAQUE` type in the database, there is a corresponding Java type.

- Use classes that implement the standard `SQLData` interface, as described in the JDBC specification.

  JPublisher generates classes for SQL object types that implement the `java.sql.SQLData` interface. When you use the `SQLData` interface, all object reference types are represented generically as `java.sql.Ref` and all collection types are represented generically as `java.sql.Array`. In addition, when using `SQLData`, there is no mechanism for representing `OPAQUE` types.

- Use `oracle.sql.*` classes.

  You can use the `oracle.sql.*` classes to represent user-defined types generically. The `oracle.sql.STRUCT` class represents all object types, the `oracle.sql.ARRAY` class represents all the variable array (VARRAY) and nested table types, the `oracle.sql.REF` class represents all the object reference types, and the `oracle.sql.OPAQUE` class represents all `OPAQUE` types. These classes are immutable in the same way that `java.lang.String` is.

  Choose this option for code that processes objects, collections, references, or `OPAQUE` types in a generic way. Unlike classes implementing `ORAData` or `SQLData`, `oracle.sql.*` classes are not strongly typed.

> **Note:** You can create your own classes, but this is not recommended. If you create your own classes or generate classes for an inheritance hierarchy of object types, then your classes must be registered using a type map.

In addition to strong typing, JPublisher-generated classes that implement `ORAData` or `SQLData` have the following advantages:

- The classes are customized, rather than generic. You access attributes of an object using `get`*`XXX`*`()` and `set`*`XXX`*`()` methods named after the particular attributes of the object. Note that you must explicitly update the object in the database if there are any changes to its data.
- The classes are mutable. You can modify attributes of an object or elements of a collection. An exception is that `ORAData` classes representing object reference types are not mutable, because an object reference does not have any subcomponents that can be modified. You can, however, use the `setValue()` method of a reference object to change the database value that the reference points to.
- You can generate Java wrapper classes that are serializable or that have the `toString()` method to print out the object along with its attribute values.

Compared to classes that implement `SQLData`, classes that implement `ORAData` are fundamentally more efficient, because `ORAData` classes avoid unnecessary conversions to native Java types.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference*.

Using Strongly Typed Object References for ORAData Implementations

For Oracle `ORAData` implementations, JPublisher always generates strongly typed object reference classes, in contrast to using the weakly typed `oracle.sql.REF` class. This is to provide greater type safety and to mirror the behavior in SQL, in which object references are strongly typed. The strongly typed classes, for example, the `PersonRef` class for references to the `PERSON` object, are wrappers for the `oracle.sql.REF` class.

In these strongly typed `REF` wrappers, a `getValue()` method produces an instance of the SQL object that is referenced as of an instance of the corresponding Java class. In the case of inheritance, the method produces an instance of a subclass of the corresponding Java class.

For example, if there is a `PERSON` object type in the database with a corresponding `Person` Java class, then there will also be a `PersonRef` Java class. The `getValue()` method of the `PersonRef` class would return a `Person` instance containing the data for a `PERSON` object in the database. In addition, JPublisher also generates a static `cast()` method on the `PersonRef` class. This permits you to convert other typed references to a `PersonRef` instance.

Whenever a SQL object type has an attribute that is an object reference, the Java class corresponding to the object type would have an attribute that is an instance of a Java class corresponding to the appropriate reference type. For example, if there is a `PERSON` object with a `MANAGER REF` attribute, then the corresponding `Person` Java class will have a `ManagerRef` attribute.

Using PL/SQL Types Through JPublisher

JDBC does not support PL/SQL-specific types, such as the `BOOLEAN` type and PL/SQL `RECORD` types that are used in stored procedures or functions. JPublisher provides the following workarounds for PL/SQL types:

- JPublisher has a type map that you can use to specify the mapping for a PL/SQL type unsupported by JDBC.
- For PL/SQL `RECORD` types or indexed-by tables types, you have the choice of JPublisher automatically creating a SQL object type or SQL collection type, respectively, as an intermediate step in the mapping.

With either workaround, JPublisher creates PL/SQL conversion functions or uses predefined conversion functions that are typically found in the `SYS.SQLJUTL` package to convert between a PL/SQL type and a corresponding SQL type. The conversion functions can be used in generated Java code that calls a stored procedure directly, or JPublisher can create a wrapper function around the PL/SQL stored procedure, where the generated Java code calls the wrapper function, which calls the conversion functions. Either way, only SQL types are exposed to JDBC.

> **See Also:** "JPublisher User Type Map and Default Type Map" on page 3-5 and "Support for PL/SQL Data Types" on page 3-10

JPublisher Mapping Categories

JPublisher offers different categories of data type mappings from SQL to Java. Each type mapping option has at least two possible values: `jdbc` or `oracle`. The `-numbertypes` option has two additional alternatives: `objectjdbc` and `bigdecimal`. The following sections describe these mappings categories.

> **See Also:** Chapter 3, "Data Type and Java-to-Java Type Mappings"

JDBC Mapping

In JDBC mapping:

- Most numeric data types are mapped to Java primitive types, such as `int` and `float`.
- The `DECIMAL` and `NUMBER` type are mapped to the `java.math.BigDecimal`.
- Large object (LOB) type and other non-numeric built-in types are mapped to the standard JDBC types, such as `java.sql.Blob` and `java.sql.Timestamp`.

For object types, JPublisher generates `SQLData` classes. Because predefined data types that are Oracle extensions, such as `BFILE` and `ROWID`, do not have JDBC mappings, only the `oracle.sql.*` mapping is supported for these types.

The Java primitive types used in the JDBC mapping do not support `NULL` values and do not guard against integer overflow or floating-point loss of precision. If you are using the JDBC mapping and you attempt to call an accessor method to get an attribute of a primitive type whose value is `NULL`, then an exception is thrown. If the primitive type is `short` or `int`, then an exception is thrown if the value is too large to fit in a `short` or `int` variable.

Object JDBC Mapping

In Object JDBC mapping, most numeric data types are mapped to Java wrapper classes, such as `java.lang.Integer` and `java.lang.Float`, and `DECIMAL` and

NUMBER are mapped to java.math.BigDecimal. This differs from the JDBC mapping, which does not use primitive types.

Object JDBC is the default mapping for numeric types. When you use the Object JDBC mapping, all your returned values are objects. If you attempt to get an attribute whose value is NULL, then a NULL object is returned. The Java wrapper classes used in the Object JDBC mapping do not guard against integer overflow or floating-point loss of precision. If you call an accessor method to get an attribute that maps to java.lang.Integer, then an exception is thrown if the value is too large to fit.

BigDecimal Mapping

In BigDecimal mapping, all numeric data types are mapped to java.math.BigDecimal. This supports NULL values and large values.

Oracle Mapping

In Oracle mapping, the numeric, LOB, or other built-in types are mapped to classes in the oracle.sql package. For example, the DATE type is mapped to oracle.sql.DATE and all numeric types are mapped to oracle.sql.NUMBER. For object, collection, and object reference types, JPublisher generates ORAData classes.

Because the Oracle mapping uses no primitive types, it can represent a NULL value as a Java null in all cases. Also, it can represent the largest numeric values that can be stored in the database, because it uses the oracle.sql.NUMBER class for all numeric types.

JPublisher Input and Output

To publish database entities, JPublisher connects to the database and retrieves descriptions of SQL types, PL/SQL packages, or server-side Java classes that you specify on the command line or in an INPUT file. By default, JPublisher connects to the database by using the JDBC Oracle Call Interface (OCI) driver, which requires an Oracle client installation, including Oracle Net Services and required support files. If you do not have an Oracle client installation, then JPublisher can use the Oracle JDBC Thin driver.

JPublisher generates a Java class for each SQL type or PL/SQL package that it translates and each server-side Java class that it processes. Generated classes include code required to read and write objects in the database. When you deploy the generated JPublisher classes, your JDBC driver installation includes all the necessary run-time files. If JPublisher generates wrapper methods for stored procedures, then the classes that it produces use the SQLJ run time during execution. In this case, you must also have the SQLJ run-time library runtime12.jar.

When you call a wrapper method on an instance of a class that was generated for a SQL object, the SQL value for the corresponding object is sent to the server along with any IN or IN OUT arguments. Then the method is invoked, and the new object value is returned to the client along with any OUT or IN OUT arguments. Note that this results in a database round trip. If the method call only performs a simple state change on the object, then there will be better performance if you write and use equivalent Java that affects the state change locally.

The number of classes that JPublisher produces depends on whether you request ORAData classes or SQLData classes.

To publish external Web services for access from inside a database, JPublisher accesses a specified Web Service Description Language (WSDL) document and directs the

generation of appropriate client proxy classes. It then generates wrapper classes, as necessary, and PL/SQL wrappers to allow Web services call-outs from PL/SQL.

The following subsections provide more detail:

- Input to JPublisher
- Output from JPublisher

See Also: "Overview of the Publishing Process: Generation and Use of Output" on page 1-20

Input to JPublisher

You can specify input options on the command line and in a JPublisher properties file. In addition to producing Java classes for the translated entities, JPublisher writes the names of the translated objects and packages to the standard output.

You can use a file known as the JPublisher `INPUT` file to specify the SQL types, PL/SQL packages, or server-side Java classes that JPublisher should publish. It also controls the naming of the generated packages and classes.

To use a properties file to specify option settings, specify the name of the properties file on the command line by using the `-props` option. JPublisher processes a properties file as if its contents were inserted in sequence on the command line at the point of the `-props` option. For additional flexibility, properties files can also be SQL script files in which the JPublisher directives are embedded in SQL comments.

See Also: "JPublisher Options" on page 6-1, "INPUT File Structure and Syntax" on page 6-52, and "Properties File Structure and Syntax" on page 6-51

Output from JPublisher

This section describes JPublisher output for user-defined object types, user-defined collection types, `OPAQUE` types, PL/SQL packages, server-side Java classes, SQL queries or DML statements, and AQs and streams.

Note: Be aware that when JPublisher publishes a database entity, such as a SQL type or PL/SQL package, it also generates classes for any types that are referenced by the entity. For example, if a stored procedure in a PL/SQL package that is being published uses a SQL object type as an argument, then a class will be generated to map to that SQL object type.

Java Output for User-Defined Object Types

For a user-defined object type, when you run JPublisher and request `ORAData` classes, JPublisher creates the following:

- An object class that represents instances of the Oracle object type in your Java program

  For each object type, JPublisher generates a *`type`*`.java` file for the class code. For example, JPublisher generates `Employee.java` for the Oracle object type `EMPLOYEE`.

- A stub subclass (optional)

It is named as specified in your JPublisher settings. You can modify the generated stub subclass for custom functionality.

- An interface for the generated class or subclass to implement (optional)
- A related reference class for object references

  JPublisher generates a *type*`Ref.java` file for the `REF` class associated with the object type. For example, JPublisher generates the `EmployeeRef.java` file for references of the Oracle object type `EMPLOYEE`.
- Java classes for any object or collection or `OPAQUE` attributes nested directly or indirectly within the top-level object

  This is necessary so that attributes can be materialized in Java whenever an instance of the top-level class is materialized. If an attribute type, such as a SQL `OPAQUE` type or a PL/SQL type, has been premapped, then JPublisher uses the target Java type from the map.

> **Notes:** For `ORAData` implementations, a strongly typed reference class is always generated, regardless of whether the SQL object type uses references.

If you request `SQLData` classes, then JPublisher does not generate the object reference class and classes for nested collection attributes or `OPAQUE` attributes.

Java Output for User-Defined Collection Types

When you run JPublisher for a user-defined collection type, you must request `ORAData` classes. JPublisher creates the following:

- A collection class to act as a type definition that corresponds to the Oracle collection type

  For each collection type JPublisher translates, it generates a *type*`.java` file. For nested tables, the generated class has methods to get and set the nested table as an entire array and to get and set individual elements of the table. JPublisher translates collection types when generating `ORAData` classes, but not when generating `SQLData` classes.
- If the elements of the collection are objects, then a Java class for the element type and Java classes for any object or collection attributes nested directly or indirectly within the element type

  This is necessary so that object elements can be materialized in Java whenever an instance of the collection is materialized.
- An interface that is implemented by the generated type (optional)

> **Note:** Unlike for object types, you do not have the option of generating user subclasses for collection types.

Java Output for OPAQUE Types

When you run JPublisher for an `OPAQUE` type, you must request `ORAData` classes. JPublisher creates a Java class that acts as a wrapper for the `OPAQUE` type, providing Java versions of the `OPAQUE` type methods as well as protected APIs to access the representation of the `OPAQUE` type in a subclass.

However, in most cases, Java wrapper classes for the SQL `OPAQUE` types are furnished by the provider of the `OPAQUE` types. For example, the `oracle.xdb.XMLType` class for the `SYS.XMLTYPE` SQL `OPAQUE` type. In such cases, ensure that the correspondence between the SQL type and the Java type is predefined to JPublisher through the type map.

Java Output for PL/SQL Packages

When you run JPublisher for a PL/SQL package, it creates a Java class with wrapper methods that invoke the stored procedures of the package on the server. `IN` arguments for the methods are transmitted from the client to the server, and `OUT` arguments and results are returned from the server to the client.

Java Output for Server-Side Java Classes and Web Services Call-Outs

When you run JPublisher for a server-side Java class used for general purposes, it creates the source code, *`type`*`.java`, for a client-side stub class that mirrors the server class. When you call the client-side methods, the corresponding server-side methods are called transparently.

For Web services call-outs, JPublisher typically generates wrapper classes for the server-side client proxy classes. These wrapper classes act as bridges to the corresponding PL/SQL wrappers. This is necessary to publish any proxy class instance method as a static method, because PL/SQL does not support instance methods.

Java Output for SQL Queries or DML Statements

When you run JPublisher for a SQL query or DML statement, it creates the following:

- A Java class that implements the method that runs the SQL statement
- A Java stub subclass, named as specified in your JPublisher settings (optional)

  You can modify this stub subclass for custom functionality.
- A Java interface for the generated class or subclass to implement (optional)

Java Output for AQs and Streams

When you run JPublisher for an AQ or a topic, it creates the following:

- A Java class for the queue or topic
- A Java class for the payload type of the queue or topic

In the case of a stream, JPublisher generates a Java class for the stream. The payload is always `SYS.ANYDATA`, which is mapped to `java.lang.Object`.

PL/SQL Output

Depending on your usage, JPublisher may generate a PL/SQL package and associated PL/SQL scripts.

PL/SQL Package

JPublisher typically generates a PL/SQL package with PL/SQL code for any of the following:

- PL/SQL call specifications for generated Java methods
- PL/SQL conversion functions and wrapper functions to support PL/SQL types
- PL/SQL table functions

Conversion functions, and optionally wrapper functions, are employed to map PL/SQL types used in the calling sequences of any stored procedures that JPublisher translates. The functions convert between PL/SQL types and corresponding SQL types, given that JDBC does not generally support PL/SQL types.

PL/SQL Scripts

JPublisher generates the following PL/SQL scripts:

- A wrapper script to create the PL/SQL package and any necessary SQL types
- A script to grant permission to run the wrapper script
- A script to revoke permission to run the wrapper script
- A script to drop the package and types created by the wrapper script

JPublisher Operation

This section discusses the following topics:

- Overview of the Publishing Process: Generation and Use of Output
- JPublisher Command-Line Syntax
- Sample JPublisher Translation

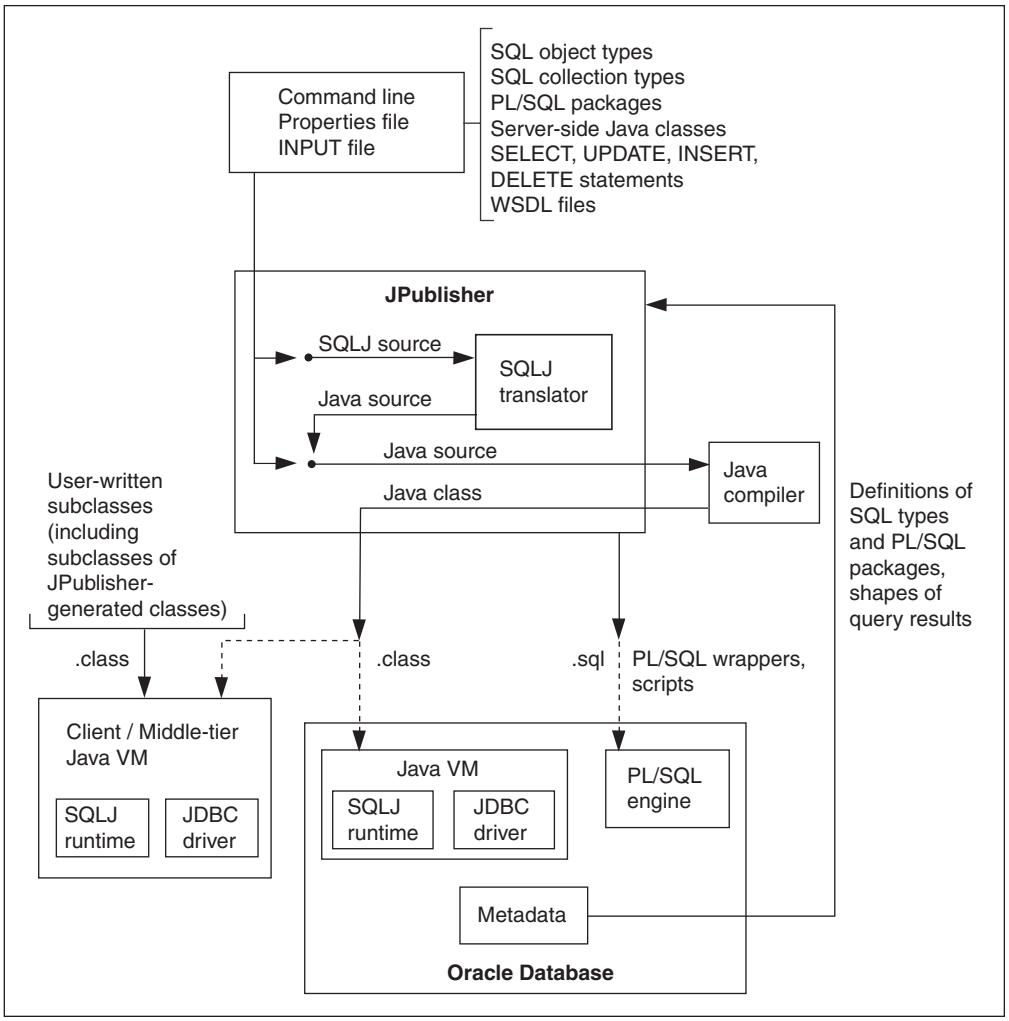
Overview of the Publishing Process: Generation and Use of Output

Figure 1–1 illustrates the basic steps for publishing specified SQL types, PL/SQL packages, or server-side Java classes. The steps are as follows:

1. Run JPublisher with input from the command line, properties file, and `INPUT` file, as desired.
2. JPublisher accesses the database to which it is attached to obtain definitions of SQL or PL/SQL entities that you specify for publishing.
3. JPublisher generates `.java` or `.sqlj` source files, as appropriate, depending on whether wrapper methods are created for stored procedures.
4. By default, JPublisher invokes the SQLJ translator, which is provided as part of JPublisher, to translate `.sqlj` files into `.java` files.
5. For SQLJ classes, by default, the SQLJ translator invokes the Java compiler to compile `.java` files into `.class` files. For non-SQLJ classes, JPublisher invokes the Java compiler.
6. JPublisher generates PL/SQL wrappers and scripts, as appropriate, in addition to the `.class` files. There is a script to create the PL/SQL wrapper package and any necessary SQL types, such as types to map to PL/SQL types, a script to drop these entities, and scripts to grant or revoke required privileges.
7. In the case of proxy class generation through the `-proxywsdl` or `-proxyclasses` option, JPublisher can load generated PL/SQL wrappers and scripts into the database to which it is connected for execution in the database PL/SQL engine.
8. By default, JPublisher loads generated Java classes for Web services call-outs into the database to which it is connected, for execution in the database JVM. JPublisher-generated classes other than those for Web services call-outs typically execute in a client or middle-tier JVM. You may also have your own classes, such

as subclasses of JPublisher-generated classes, that would typically execute in a client or middle-tier JVM.

Figure 1–1 Translating and Using JPublisher-Generated Code



JPublisher Command-Line Syntax

On most operating systems, you can start JPublisher from the command line by typing `jpub` followed by a series of option settings, as follows:

```
% jpub -option1=value1 -option2=value2 ...
```

JPublisher responds by connecting to the database and obtaining the declarations of the types or packages you specify. It then generates one or more custom Java classes and writes the names of the translated object types or PL/SQL packages to the standard output.

Here is an example of a single wraparound command that invokes JPublisher:

```
% jpub -user=scott/tiger -input=demoin -numbertypes=oracle -usertypes=oracle
       -dir=demo -d=demo -package=corp
```

Enter the command on the command line, allowing it to wrap as necessary. This command directs JPublisher to connect to the database with the user name `SCOTT` and

password TIGER and to translate data types to Java classes, based on instructions in the INPUT file demoin. The -numbertypes=oracle option directs JPublisher to map object attribute types to Java classes supplied by Oracle, and the -usertypes=oracle option directs JPublisher to generate Oracle-specific ORAData classes. JPublisher places the classes that it generates in the corp package under the demo directory.

> **Note:** This chapter refers to the input file specified by the -input option as the INPUT file to distinguish from any other kinds of input files.

JPublisher also supports specification of .java files, or .sqlj files, if you are using SQLJ source files directly, on the JPublisher command line. In addition to any JPublisher-generated files, the specified files are translated and compiled. For example:

```
% jpub ...options... Myclass.java
```

> **Notes:**
>
> - No spaces are permitted around the equal sign (=) on the JPublisher command line.
> - If you run JPublisher without any command-line input, then it displays an option list and then terminates.

Sample JPublisher Translation

This section provides a sample JPublisher translation of a user-defined object type. At this point, do not worry about the details of the code JPublisher generates. You can find more information about JPublisher input and output files, options, data type mappings, and translation later in this manual.

> **Note:** For more examples, go to *ORACLE_HOME*/sqlj/demo/jpub in your Oracle Database installation.

Create the object type EMPLOYEE:

```
CREATE TYPE employee AS OBJECT
(
    name       VARCHAR2(30),
    empno      INTEGER,
    deptno     NUMBER,
    hiredate   DATE,
    salary     REAL
);
```

The INTEGER, NUMBER, and REAL types are all stored in the database as NUMBER types, but after translation they have different representations in the Java program, based on your setting of the -numbertypes option.

Assume JPublisher translates the types according to the following command entered on the command line:

```
% jpub -user=scott/tiger -dir=demo -numbertypes=objectjdbc -builtintypes=jdbc
```

```
-package=corp -case=mixed -sql=Employee
```

See Also: "JPublisher Options" on page 6-1

Note that JPublisher generates a non-SQLJ class, because the `EMPLOYEE` object type does not define any methods.

Because `-dir=demo` and `-package=corp` are specified on the JPublisher command line, the translated class `Employee` is written to `Employee.java` at the following location:

```
./demo/corp/Employee.java
```

Note: This location is specific for a UNIX system.

The `Employee.java` class file would contain the code shown in the following example.

Note: The details of the code JPublisher generates are subject to change. In particular, non-public methods, non-public fields, and all method bodies may be generated differently.

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Employee implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.EMPLOYEE";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  private static int[] _sqlType =  { 12,4,2,91,7 };
  private static ORADataFactory[] _factory = new ORADataFactory[5];
  protected static final Employee _EmployeeFactory = new Employee(false);

  public static ORADataFactory getORADataFactory()
  { return _EmployeeFactory; }

  /* constructor */
  protected Employee(boolean init)
  { if(init) _struct = new MutableStruct(new Object[5], _sqlType, _factory); }
  public Employee()
  { this(true); }
  public Employee(String name, Integer empno, java.math.BigDecimal deptno,
                  java.sql.Timestamp hiredate, Float salary)
   throws SQLException
```

```
  { this(true);
    setName(name);
    setEmpno(empno);
    setDeptno(deptno);
    setHiredate(hiredate);
    setSalary(salary);
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _struct.toDatum(c, _SQL_NAME);
  }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { return create(null, d, sqlType); }
  protected ORAData create(Employee o, Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    if (o == null) o = new Employee(false);
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
    return o;
  }
  /* accessor methods */
  public String getName() throws SQLException
  { return (String) _struct.getAttribute(0); }

  public void setName(String name) throws SQLException
  { _struct.setAttribute(0, name); }

  public Integer getEmpno() throws SQLException
  { return (Integer) _struct.getAttribute(1); }

  public void setEmpno(Integer empno) throws SQLException
  { _struct.setAttribute(1, empno); }

  public java.math.BigDecimal getDeptno() throws SQLException
  { return (java.math.BigDecimal) _struct.getAttribute(2); }

  public void setDeptno(java.math.BigDecimal deptno) throws SQLException
  { _struct.setAttribute(2, deptno); }

  public java.sql.Timestamp getHiredate() throws SQLException
  { return (java.sql.Timestamp) _struct.getAttribute(3); }

  public void setHiredate(java.sql.Timestamp hiredate) throws SQLException
  { _struct.setAttribute(3, hiredate); }

  public Float getSalary() throws SQLException
  { return (Float) _struct.getAttribute(4); }

  public void setSalary(Float salary) throws SQLException
  { _struct.setAttribute(4, salary); }

}
```

Code Generation Notes

- JPublisher also generates object constructors based on the object attributes.

- Additional private or public methods may be generated with other option settings. For example, the `-serializable=true` setting results in the object wrapper class implementing the interface `java.io.Serializable` and in the generation of private `writeObject()` and `readObject()` methods. In addition, the `-tostring=true` setting results in the generation of a public `toString()` method.
- There is a protected `_struct` field in JPublisher-generated code for SQL object types. This is an instance of the `oracle.jpub.runtime.MutableStruct` internal class. It contains the data in original SQL format. In general, you should never reference this field directly. Instead, use the `-methods=always` or `-methods=named` setting, as necessary, to ensure that JPublisher produces `setFrom()` and `setValueFrom()` methods, and then use these methods when extending a class.

  See Also: "The setFrom(), setValueFrom(), and setContextFrom() Methods" on page 4-11

- JPublisher generates SQLJ classes instead of non-SQLJ classes in the following circumstances:
  - The SQL object being published has methods, and the `-methods=false` setting is not specified.
  - A PL/SQL package, stored procedure, query, or DML statement is published, and the `-methods=false` setting is not specified.

  In addition:

  - If a SQLJ class is created for a type definition, then a SQLJ class is also created for the corresponding REF definition.
  - If a SQLJ class is created for a base class, then SQLJ classes are also created for any subclasses.

  This means that, in a backward-compatibility mode, JPublisher generates `.sqlj` files instead of `.java` files.

  Note: The JPublisher version provided with Oracle8*i* Database generates implementations of the now-deprecated `CustomDatum` and `CustomDatumFactory` interfaces, instead of `ORAData` and `ORADataFactory`. In fact, it is still possible to do this through the JPublisher `-compatible` option. This is required if you are using an Oracle8*i* JDBC driver.

JPublisher also generates an `EmployeeRef.java` class. The source code is as follows:

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class EmployeeRef implements ORAData, ORADataFactory
```

```
{
  public static final String _SQL_BASETYPE = "SCOTT.EMPLOYEE";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;

private static final EmployeeRef _EmployeeRefFactory = new EmployeeRef();

  public static ORADataFactory getORADataFactory()
  { return _EmployeeRefFactory; }
  /* constructor */
  public EmployeeRef()
  {
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _ref;
  }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    EmployeeRef r = new EmployeeRef();
    r._ref = (REF) d;
    return r;
  }

  public static EmployeeRef cast(ORAData o) throws SQLException
  {
     if (o == null) return null;
     try { return (EmployeeRef) getORADataFactory().create(o.toDatum(null),
           OracleTypes.REF); }
     catch (Exception exn)
     { throw new SQLException("Unable to convert "+o.getClass().getName()+" to
             EmployeeRef: "+exn.toString()); }
  }

  public Employee getValue() throws SQLException
  {
     return (Employee) Employee.getORADataFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

  public void setValue(Employee c) throws SQLException
  {
    _ref.setValue((STRUCT) c.toDatum(_ref.getJavaSqlConnection()));
  }
}
```

Note: JPublisher also generates a public static `cast()` method to cast from other strongly typed references into a strongly typed reference instance.

2

Using JPublisher

This chapter describes how you can use JPublisher for:

- Publishing User-Defined SQL Types
- Publishing PL/SQL Packages
- Publishing Oracle Streams AQ
- Publishing Server-Side Java Classes Through Native Java Interface
- Publishing Server-Side Java Classes Through PL/SQL Wrappers
- Publishing Server-Side Java Classes to PL/SQL
- Publishing Server-Side Java Classes to Table Functions
- Publishing Web Services Client into PL/SQL

Publishing User-Defined SQL Types

Using JPublisher to publish SQL objects or collections as Java classes is straightforward. This section provides examples of this for the Order Entry (`OE`) schema, which is part of the Oracle Database sample schema. If you do not have the sample schema installed, but have your own object types that you would like to publish, then replace the user name, password, and object names accordingly.

Assuming that the password for the `OE` schema is `OE`, use the following command to publish the `CATEGORY_TYP` SQL object type, where `%` is the system prompt:

```
% jpub -user=OE/OE -sql=CATEGORY_TYP:CategoryTyp
```

The JPublisher `-user` option specifies the user name and password. The `-sql` option specifies the types to be published. The SQL type and Java class is separated by a colon (`:`). `CATEGORY_TYP` is the name of the SQL type, and `CategoryTyp` is the name of the corresponding Java class that is to be generated.

> **See Also:** "Declaration of Object Types and Packages to Translate" on page 6-14.

JPublisher echoes the names of the SQL types that it publishes to the standard output:

```
OE.CATEGORY_TYP
```

In addition to the `CategoryTyp.java` file, JPublisher also generates the `CategoryTypeRef.java` file. This is a strongly typed wrapper class for SQL object references to `OE.CATEGORY_TYP`. Both these files can be compiled with the Java compiler, `javac`.

Another example of publishing SQL object types, in this case the CUSTOMER_TYP type, by using the shorthand -u for "-user=" and -s for "-sql=" is:

```
% jpub -u OE/OE -s CUSTOMER_TYP:CustomerTyp
```

The options -u and -s are followed by a space and then the value.

JPublisher reports a list of SQL object types. Whenever it encounters an object type for the first time, whether through an attribute, an object reference, or a collection that has element types as objects or collections, it automatically generates a wrapper class for that type as well. The list of SQL object types for the OE schema are:

```
OE.CUSTOMER_TYP
OE.CORPORATE_CUSTOMER_TYP
OE.CUST_ADDRESS_TYP
OE.PHONE_LIST_TYP
OE.ORDER_LIST_TYP
OE.ORDER_TYP
OE.ORDER_ITEM_LIST_TYP
OE.ORDER_ITEM_TYP
OE.PRODUCT_INFORMATION_TYP
OE.INVENTORY_LIST_TYP
OE.INVENTORY_TYP
OE.WAREHOUSE_TYP
```

Two source files are generated for each object type in this example: one for a Java class, such as CustomerTyp, to represent instances of the object type, and one for a reference class, such as CustomerTypeRef, to represent references to the object type.

Notice the naming scheme that JPublisher uses by default. For example, the OE.PRODUCT_INFORMATION_TYP SQL type is converted to a Java class, ProductInformationTyp.

Although JPublisher automatically generates wrapper classes for embedded types, it does not do so for subtypes of given object types. In this case, you have to explicitly enumerate all the subtypes that you want to have published. The CATEGORY_TYP type has three subtypes: LEAF_CATEGORY_TYP, COMPOSITE_CATEGORY_TYP, and CATALOG_TYP. The following is a single, wraparound JPublisher command line to publish the subtypes of the object type:

```
% jpub  -u OE/OE  -s COMPOSITE_CATEGORY_TYP:CompositeCategoryTyp
        -s LEAF_CATEGORY_TYP:LeafCategoryTyp,CATALOG_TYP:CatalogTyp
```

JPublisher lists the processed types as output, as follows:

```
OE.COMPOSITE_CATEGORY_TYP
OE.SUBCATEGORY_REF_LIST_TYP
OE.LEAF_CATEGORY_TYP
OE.CATALOG_TYP
OE.CATEGORY_TYP
OE.PRODUCT_REF_LIST_TYP
```

Keep in mind the following information:

- If you want to unparse several types, then you can list them all together in the -sql or -s option, each separated by a comma, or you can supply several -sql options on the command line.
- Although JPublisher does not automatically generate wrapper classes for all subtypes, it *does* generate them for all supertypes.

- For SQL objects with methods, such as `CATALOG_TYP`, JPublisher uses SQLJ classes to implement the wrapper methods. In Oracle Database 10*g*, the use of SQLJ classes, as opposed to regular Java classes, is invisible to you unless you use one of the backward-compatibility modes.

  > **Note:** Prior to Oracle Database 10*g*, the generation of SQLJ classes resulted in the creation of visible `.sqlj` source files. In Oracle Database 10*g*, if you set the JPublisher `-compatible` flag to a value of `8i`, `both8i`, `9i`, or `sqlj`, then visible `.sqlj` source files will be generated.
  >
  > In any of these modes, you can use the JPublisher `-sqlj` option as an alternative to using the `sqlj` command-line utility to translate `.sqlj` files.

If the code that JPublisher generates does not provide the functionality or behavior you want, then you can extend generated wrapper classes to override or complement their functionality. Consider the following example:

```
% jpub -u OE/OE -s WAREHOUSE_TYP:JPubWarehouse:MyWarehouse
```

The JPublisher output is:

```
OE.WAREHOUSE_TYP
```

With this command, JPublisher generates both `JPubWarehouse.java` and `MyWarehouse.java`. The `JPubWarehouse.java` file is regenerated every time you rerun this command. The `MyWarehouse.java` generated file can be customized by you and will not be overwritten by future runs of this command. You can add new methods in `MyWarehouse.java` and override the method implementations from `JPubWarehouse.java`.

The class that is used to materialize `WAREHOUSE_TYP` instances in Java is the specialized `MyWarehouse` class. If you want user-specific subclasses for all types in an object type hierarchy, then you must specify *triplets* of the form *`SQL_TYPE`*`:`*`JPubClass`*`:`*`UserClass`*, for all members of the hierarchy, as shown in the preceding JPublisher command.

Once you have generated and compiled Java wrapper classes with JPublisher, you can use the object wrappers directly.

> **Note:** The preceding examples using the `OE` schema are for illustrative purposes only and may not be completely up-to-date regarding the composition of the schema.

The following SQLJ class calls a PL/SQL stored procedure. Assume that `register_warehouse` takes a `WAREHOUSE_TYP` instance as an `IN OUT` parameter. Code comments show the corresponding `#sql` command. By default, JPublisher generates and translates the SQLJ code automatically.

```
java.math.BigDecimal location = new java.math.BigDecimal(10);
java.math.BigDecimal warehouseId = new java.math.BigDecimal(10);
MyWarehouse w = new MyWarehouse(warehouseId,"Industrial Park",location);
//  ************************************************************
//  #sql { call register_warehouse(:INOUT w) };
//  ************************************************************
//
```

```
// declare temps
oracle.jdbc.OracleCallableStatement __sJT_st = null;
sqlj.runtime.ref.DefaultContext __sJT_cc =
     sqlj.runtime.ref.DefaultContext.getDefaultContext();
if (__sJT_cc==null)
  sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();
sqlj.runtime.ExecutionContext.OracleContext __sJT_ec =
     ((__sJT_cc.getExecutionContext()==null) ?
          sqlj.runtime.ExecutionContext.raiseNullExecCtx() :
               __sJT_cc.getExecutionContext().getOracleContext());
try
{
  String theSqlTS = "BEGIN register_warehouse( :1 ) \n; END;";
  __sJT_st = __sJT_ec.prepareOracleCall(__sJT_cc,"0RegisterWarehouse",theSqlTS);
  if (__sJT_ec.isNew())
  {
    __sJT_st.registerOutParameter(1,2002,"OE.WAREHOUSE_TYP");
  }
  // set IN parameters
  if (w==null)
    __sJT_st.setNull(1,2002,"OE.WAREHOUSE_TYP");
  else __sJT_st.setORAData(1,w);
    // execute statement
    __sJT_ec.oracleExecuteUpdate();
  // retrieve OUT parameters
  w = (MyWarehouse)__sJT_st.getORAData(1,MyWarehouse.getORADataFactory());
}
finally
{
  __sJT_ec.oracleClose();
}
```

In Java Database Connectivity (JDBC), you typically register the relationship between the SQL type name and the corresponding Java class in the type map for your connection instance. This is required once for each connection. This type mapping can be done as shown in the following example:

```
java.util.Map typeMap = conn.getTypeMap();
typeMap.put("OE.WAREHOUSE_TYP", MyWarehouse.class);
conn.setTypeMap(typeMap);
```

The following JDBC code is equivalent to the JPublisher output, that is, the translated SQLJ code, shown previously:

```
CallableStatement cs = conn.prepareCall("{call register_warehouse(?)}");
((OracleCallableStatement)cs).registerOutParameter
                (1,oracle.jdbc.OracleTypes.STRUCT,"OE.WAREHOUSE_TYP");
cs.setObject(w);
cs.executeUpdate();
w = cs.getObject(1);
```

See Also: "Publishing PL/SQL Packages" on page 2-4

Publishing PL/SQL Packages

In addition to mapping SQL objects, you may want to encapsulate entire PL/SQL packages as Java classes. JPublisher offers functionality to create Java wrapper methods for the stored procedures of a PL/SQL package.

However, the concept of representing PL/SQL stored procedures as Java methods presents a problem. Arguments to the PL/SQL functions and procedures may use the PL/SQL `OUT` or `IN OUT` mode, but there are no equivalent modes for passing arguments in Java. A method that takes an `int` argument, for example, is not able to modify this argument in such a way that its callers can receive a new value for it. As a workaround, JPublisher can generate single-element arrays for `OUT` and `IN OUT` arguments. For example, consider an integer array `int[] abc`. The input value is provided in `abc[0]`, and the modified output value is also returned in `abc[0]`. JPublisher also uses a similar pattern when generating code for SQL object type methods.

> **Note:** If your stored procedures use types that are specific to PL/SQL and are not supported by Java, then special steps are required to map these arguments to SQL and then to Java.

The following command publishes the `SYS.DBMS_LOB` package into Java:

```
% jpub  -u SCOTT/TIGER  -s SYS.DBMS_LOB:DbmsLob
```

The JPublisher output is:

```
SYS.DBMS_LOB
```

Because `DBMS_LOB` is publicly visible, you can access it from a different schema, such as `SCOTT`. Note that this JPublisher invocation creates a SQLJ class in `DbmsLob.java` that contains the calls to the PL/SQL package. The generated Java methods are actually the instance methods. The idea is that you create an instance of the package using a JDBC connection or a SQLJ connection context and then call the methods on that instance.

> **See Also:** "Treatment of Output Parameters" on page 4-1 and "Support for PL/SQL Data Types" on page 3-10

Use of Object Types Instead of Java Primitive Numbers

When you examine the generated code, notice that JPublisher has generated `java.lang.Integer` as arguments to various methods. Using Java object types, such as `Integer`, instead of Java primitive types, such as `int`, permits you to represent SQL `NULL` values directly as Java `null`s, and JPublisher generates these by default. However, for the `DBMS_LOB` package, `int` is preferable over the `Integer` object type. The following modified JPublisher invocation accomplishes this through the `-numbertypes` option:

```
% jpub -numbertypes=jdbc  -u SCOTT/TIGER  -s SYS.DBMS_LOB:DbmsLob
```

The JPublisher output is:

```
SYS.DBMS_LOB
```

> **See Also:** "Mappings for Numeric Types" on page 6-23

Wrapper Class for Procedures at the SQL Top Level

JPublisher also enables you to generate a wrapper class for the functions and procedures at the SQL top level. Use the special package name `TOPLEVEL`, as in the following example:

```
% jpub  -u SCOTT/TIGER  -s TOPLEVEL:SQLTopLevel
```

The JPublisher output is:

```
SCOTT.top-level_scope
```

A warning appears if there are no stored functions or procedures in the SQL top-level scope.

Publishing Oracle Streams AQ

Publishing Oracle Streams Advanced Queue (AQ) as Java classes is similar to publishing PL/SQL stored procedures. JPublisher exposes a queue as a Java program using AQ Java Message Service (JMS) application programming interfaces (APIs). This Java program can be further published into Web services by the Web services assembler. You can perform the following:

- Publishing a Queue as a Java Class
- Publishing a Topic as a Java Class
- Publishing a Stream as a Java Class

Oracle Streams AQ can be categorized into queue, topic, and stream. A queue is a one-to-one message channel with a declared payload type. A topic is a one to many message channel with a declared payload type. A stream is a queue or topic with `SYS.ANYDATA` as the payload type.

You can publish a queue, topic, or stream using the `-sql` option as follows:

```
%jpub -user=SCOTT/TIGER -sql=AQNAME:javaName
```

`AQNAME` is the name of a queue table, queue, topic, or stream. *`javaName`* is the name of the corresponding Java class.

In Microsoft Windows, you need to add the following Java Archive (JAR) files to `CLASSPATH` for JPublisher to publish a queue. These two files are required for the running of the JPublisher-generated code for Oracle Streams AQ.

```
ORACLE_HOME/rdbms/jlib/jmscommon.jar
ORACLE_HOME/rdbms/jlib/aqapi.jar
```

On UNIX, the `jpub` script distributed with Oracle Database 10*g* release 2 (10.2) includes these JAR files.

For Oracle Streams AQ, the usage of the `-sql` option is the same as SQL types and PL/SQL stored procedures. You can specify subclasses and interfaces. Other options available to SQL types and PL/SQL packages, such as `-genpattern`, `-style`, `-builtintypes`, and `-compatible`, are also available with Oracle Streams AQ.

Publishing a Queue as a Java Class

You can publish a queue using the same settings that are used for publishing a SQL type or PL/SQL stored procedure.

Consider a queue, `toy_queue`, declared as follows:

```
CREATE TYPE scott.queue_message AS OBJECT (
  Subject VARCHAR2(30),
  Text VARCHAR2(80)
);
dbms_aqadm.create_queue_table (
  Queue_table => 'scott.queue_queue_table',
```

```
  Queue_payload_type => 'scott.queue_message'
);
dbms_aqadm.create_queue (
  queue_name  => 'scott.toy_queue',
  queue_table => 'scott.queue_queue_table'
);
dbms_aqadm.start_queue (
  queue_name => 'scott.toy_queue'
);
```

The following command publishes `toy_queue` as a Java program:

```
% jpub  -user=SCOTT/TIGER  -sql=toy_queue:ToyQueue
```

> **Note:** When creating a queue or topic, you can specify a SQL type as the payload type. The payload type is transformed into and from the JMS message types.

The command generates `ToyQueue.java`, with the following APIs:

```
public class ToyQueue
{
   public ToyQueue();
   public ToyQueue(java.sql.Connection conn);
   public ToyQueue(javax.sql.DataSource dataSource);
   public void setConnection(java.sql.Connection conn);
   public void setDataSource(javax.sql.DataSource ds);
   public void addTypeMap(String sqlName, String javaName);
   public void send(QueueMessage payload);
   public QueueMessage receive();
   public QueueMessage receiveNoWait();
   public QueueMessage receive(java.lang.String selector, boolean noWait);
}
```

Like for PL/SQL stored procedures, JPublisher generates connection and data source management APIs, such as `setConnection()` and `setDataSource()`. The `addTypeMap()` method enables you to specify type mapping if the payload type is a SQL type hierarchy. The `send()` method enqueues a message. The `receive()` method dequeues a message from the queue. This method blocks until a message is available to dequeue. The `receiveNoWait()` method dequeues a message and returns `null` if no message is available. The last `receive()` method in the `ToyQueue` class dequeues a message satisfying the selector. The selector is a condition specified in the AQ convention. For example, consider the condition:

```
priority > 3 and Subject IN ('spider','tank')
```

This selects messages with priority higher than `3` and with `spider` and `tank` as the `Subject` attribute.

`QueueMessage` is a subclass of `ORAData` and is generated for the `queue_message` payload type, which is a SQL type published as the result of publishing the queue.

The following sample client code uses the generated `ToyQueue` class. The client code sends a message to the queue, dequeues the queue using the block operator `receive()`, and continues dequeuing messages using `receiveNoWait()`, until all messages in the queue are dequeued.

```
...
```

```
ToyQueue q = new ToyQueue(getConnection());
QueueMessage m = new QueueMessage("scooby doo", "lights out");
q.send(m);
System.out.println("Message sent: " + m.getSubject() + " " + m.getText());
m = new QueueMessage("dalmatian", "solve the puzzle");
q.send(m);
System.out.println("Message sent: " + m.getSubject() + " " + m.getText());
m = q.receive();
while (m!=null)
{
  System.out.println("Message received: " + m.getSubject() + " " + m.getText());
  m = q.receiveNoWait();
}
...
```

Publishing a Topic as a Java Class

Consider a topic declared as follows:

```
CREATE TYPE scott.topic_message AS OBJECT (
  Subject VARCHAR2(30),
  Text VARCHAR2(80)
);
dbms_aqadm.create_queue_table (
  Queue_table => 'scott.topic_queue_table',
  Multiple_consumers => TRUE,
  Queue_payload_type => 'scott.topic_message'
);
dbms_aqadm.create_queue (
  queue_name  => 'scott.toy_topic',
  queue_table => 'scott.topic_queue_table'
);
dbms_aqadm.start_queue (
  queue_name => 'scott.toy_topic'
);
```

The queue table, `topic_queue_table`, has the `Multiple_consumers` property set to `TRUE`, indicating that the queue table hosts topics instead of queues.

You can publish the topic as follows:

```
% jpub -user=SCOTT/TIGER -sql=toy_topic:ToyTopic
```

The command generates `ToyTopic.java` with the following APIs:

```
public class ToyTopic
{
  public ToyTopic(javax.sql.DataSource dataSource);
  public void setConnection(java.sql.Connection conn);
  public void setDataSource(javax.sql.DataSource ds);
  public void addTypeMap(String sqlName,String javaName);
  public void publish(TopicMessage payload);
  public void publish(TopicMessage payload, java.lang.String[] recipients);
  public void publish(TopicMessage payload, int deliveryMode, int priority,
        long timeToLive);
  public void subscribe(java.lang.String subscriber);
  public void unsubscribe(java.lang.String subscriber);
  public TopicMessage receiveNoWait(java.lang.String receiver);
  public TopicMessage receive(java.lang.String receiver);
  public TopicMessage receive(java.lang.String receiver,
```

```
        java.lang.String selector);
}
```

The `publish` methods enqueue a message addressed to all the subscribers or a list of subscribers. The `deleveryMode` parameter takes the value `javax.jms.DeliveryMode.PERSISTENT` or `javax.jms.DeliveryMode.NON_PERSISTENT`. However, only `DeliveryMode.PERSISTENT` is supported in Oracle Database 10*g* release 2 (10.2). The `priority` parameter specifies the priority of the message. The `timeToLive` parameter specifies the time in milliseconds after which the message will be timed out. A value of `0` indicates the message is not timed out.

The `receive` methods dequeue a message addressed to the specified receiver.

The following sample client code uses the generated `ToyTopic` class. The client sends a message to two receivers, `ToyParty` and `ToyFactory`, and then dequeues the topic as `ToyParty`, `ToyLand`, and `ToyFactory` respectively.

```
...
ToyTopic topic = new ToyTopic(getConnection());
TopicMessage m = new TopicMessage("scooby doo", "lights out");

topic.publish(m, new String[]{"ToyParty", "ToyFactory"});
System.out.println("Message broadcasted: " + m.getSubject() + " " + m.getText());
m = new TopicMessage("dalmatian", "solve the puzzle");
topic.publish(m, new String[]{"ToyParty", "ToyLand"});
System.out.println("Message broadcasted: " + m.getSubject() + " " + m.getText());

m = topic.receive("ToyParty");
System.out.println("ToyParty receive " + m.getSubject() + " " + m.getText());
m = topic.receive("ToyParty");
System.out.println("ToyParty receive " + m.getSubject() + " " + m.getText());

m = topic.receiveNoWait("ToyLand");
System.out.println("ToyFactory receive " + m.getSubject() + " " + m.getText());
m = topic.receiveNoWait("ToyFactory");
System.out.println("ToyFactory receive " + m.getSubject() + " " + m.getText());
m = topic.receiveNoWait("ToyFactory");
...
```

Publishing a Stream as a Java Class

A stream is a special case of AQ. It can have only `SYS.ANYDATA` as the payload type. As a limitation, JPublisher-generated code for streams requires the JDBC Oracle Call Interface (OCI) driver. However, the code generated for queue and topic run on both the JDBC Thin and JDBC OCI driver.

Publishing a stream is similar to publishing an AQ. The following command will publish the stream, `toy_stream`:

```
% jpub  -user=SCOTT/TIGER  -sql=toy_stream:ToyStream
```

This command generates the `ToyStream.java` file.

The difference between publishing a stream and an AQ or a topic is that when a stream is published, the payload type will always be `SYS.ANYDATA`, which is mapped to `java.lang.Object`.

The `ToyStream.java` file contains the following APIs:

```
public class ToyStream
{
```

```
  public ToyStream();
  public ToyStream(java.sql.Connection conn);
  public ToyStream(javax.sql.DataSource dataSource);
  public void setConnection(java.sql.Connection conn);
  public void setDataSource(javax.sql.DataSource ds);
  public void addTypeMap(String sqlName, String javaName);
  public void publish(Object payload);
  public void publish(Object payload, java.lang.String[] recipients);
  public void publish(Object payload, int deliveryMode,
         int priority, long timeToLive);
  public void subscribe(java.lang.String subscriber);
  public void unsubscribe(java.lang.String subscriber);
  public Object receiveNoWait(java.lang.String receiver);
  public Object receive(java.lang.String receiver);
  public Object receive(java.lang.String receiver, java.lang.String selector);
  public Object receive(java.lang.String receiver, java.lang.String selector,
         long timeout);
}
```

Here is a sample code that uses the generated `ToyStream` class:

```
...
System.out.println("*** testStream with an OCI connection");
Object response = null;
ToyStream stream = new ToyStream(getOCIConnection());

stream.publish("Seaside news", new String[]{"ToyParty"});
response = stream.receive("ToyParty");
System.out.println("Received: " + response);

stream.publish(new Integer(333), new String[]{"ToyParty"});
response = stream.receive("ToyParty");
System.out.println("Received: " + response);

stream.publish(new Float(3.33), new String[]{"ToyParty"});
response = stream.receive("ToyParty");
System.out.println("Received: " + response);

stream.publish("Science Monitor".getBytes(), new String[]{"ToyParty"});
response = stream.receive("ToyParty");
System.out.println("Received: " + new String((byte[])response));

stream.publish(new String[]{"gamma", "beta"}, new String[]{"ToyParty"});
response = stream.receive("ToyParty");
System.out.println("Received: " + ((String[]) response)[0]);

HashMap map = new HashMap();
map.put("US", "dollar");
map.put("Japan", "yen");
map.put("Austrilia", "dollar");
map.put("Britian", "pound");
stream.publish(map, new String[]{"ToyParty"});
response = stream.receive("ToyParty");
map = (HashMap) response;
System.out.println("Message received: " + map.get("Britian") + ", " +
map.get("US") + ", " + map.get("Austrilia"));

stream.addTypeMap("SCOTT.QUEUE_MESSAGE", "queue.wrapper.simple.QueueMessage");
stream.addTypeMap("QUEUE_MESSAGE", "queue.wrapper.simple.QueueMessage");
QueueMessage m = new QueueMessage("Knowing", "world currency");
stream.publish(m, new String[]{"ToyParty"});
```

```
response = stream.receive("ToyParty");
System.out.println(response);
m = (QueueMessage) response;
System.out.println("Message received: " + m.getSubject() + " " + m.getText());
...
```

The sample code sends messages of various types, such as `String`, `Integer`, and `java.util.Map`. For the `QueueMessage` JDBC custom type, the `addTypeMap()` method is called to specify SQL type to Java type mapping.

Publishing Server-Side Java Classes Through Native Java Interface

Oracle Database 10*g* introduces the native Java interface feature for calls to server-side Java code. Prior to Oracle Database 10*g*, calling Java stored procedures and functions from a database client required JDBC calls to the associated PL/SQL wrappers. Each PL/SQL wrapper had to be manually published with a SQL signature and a Java implementation. This process had the following disadvantages:

- The signatures permitted only Java types that had direct SQL equivalents.
- Exceptions issued in Java were not properly returned.

The JPublisher `-java` option provides functionality to overcome these disadvantages.

To remedy the deficiencies of JDBC calls to associated PL/SQL wrappers, the `-java` option makes use of an API for direct invocation of static Java methods. This functionality is also useful for Web services.

The functionality of the `-java` option mirrors that of the `-sql` option, creating a client-side Java stub class to access a server-side Java class. This is in contrast to creating a client-side Java class to access a server-side SQL object or PL/SQL package. The client-side stub class uses SQL code that mirrors the server-side class and includes the following features:

- Methods corresponding to the public, static methods of the server class
- Two constructors, one that takes a JDBC connection and one that takes the SQLJ default connection context instance

At run time, the stub class is instantiated with a JDBC connection. Calls to its methods result in calls to the corresponding methods of the server-side class. Any Java types used in these published methods must be primitive or serializable.

You can use the `-java` option to publish a server-side Java class, as follows:

```
-java=className
```

Consider the `oracle.sqlj.checker.JdbcVersion` server-side Java class, with the following APIs:

```
public class oracle.sqlj.checker.JdbcVersion
{
  public oracle.sqlj.checker.JdbcVersion();
  public static int getDriverMajorVersion();
  public static int getDriverMinorVersion();
  public static java.lang.String getDriverName();
  public static java.lang.String getDriverVersion();
  public static java.lang.String getJdbcLibraryName();
  public static java.lang.String getRecommendedRuntimeZip();
  public static java.lang.String getRuntimeVersion();
  public static java.lang.String getSqljLibraryName();
  public static boolean hasNewStatementCache();
```

```
  public static boolean hasOracleContextIsNew();
  public static boolean hasOracleSavepoint();
  public static void main(java.lang.String[]);
  public java.lang.String toString();
  public static java.lang.String to_string();
}
```

As an example, assume that you want to call the following method on the server:

```
public String oracle.sqlj.checker.JdbcVersion.to_string();
```

Use the following command to publish `JdbcVersion` for client-side invocation:

```
% jpub -sql=scott/tiger -java=oracle.sqlj.checker.JdbcVersion:JdbcVersion Client
```

This command generates the client-side Java class, `JdbcVersionClient`, which contains the following APIs:

```
public class JdbcVersionClient
{
  public long newInstance();
  public JdbcVersionClient();
  public JdbcVersionClient(java.sql.Connection conn);
  public JdbcVersionClient(sqlj.runtime.ref.DefaultContext ctx);
  public java.lang.String toString(long _handle);
  public int getDriverMajorVersion();
  public int getDriverMinorVersion();
  public java.lang.String getDriverName();
  public java.lang.String getDriverVersion();
  public java.lang.String getJdbcLibraryName();
  public java.lang.String getRecommendedRuntimeZip();
  public java.lang.String getRuntimeVersion();
  public java.lang.String getSqljLibraryName();
  public boolean hasNewStatementCache();
  public boolean hasOracleContextIsNew();
  public boolean hasOracleSavepoint();
  public void main(java.lang.String[] p0);
  public java.lang.String to_string();
}
```

Compare `oracle.sqlj.checker.JdbcVersion` with `JdbcVersionClient`. All static methods are mapped to instance methods in the client-side code. A instance method in the server-side class, `toString()` for example, is mapped to a method with an extra handle. A handle represents an instance of `oracle.sqlj.checker.JdbcVersion` in the server. The handle is used to call the instance method on the server-side. The extra method in `JdbcVersionClient` is `newInstance()`, which creates a new instance of `oracle.sqlj.checker.JdbcVersion` in the server and returns its handle.

Publishing the server-side Java class has the following constraints:

- Instance methods can be published only if the class to be published has a `public` empty constructor.
- Only serializable parameter and return types are supported. Methods with nonserializable types will not be published.
- Oracle Database 10*g* is required, or `sqljutl.jar` needs to be installed in `SYS`.

See Also: "Declaration of Server-Side Java Classes to Publish" on page 6-7

Publishing Server-Side Java Classes Through PL/SQL Wrappers

In Oracle Database 10*g* release 2 (10.2), JPublisher provides a new approach to publish server-side Java classes. It generates the following to call server-side Java:

- Java stored procedure wrapper for the server-side class
- PL/SQL wrapper for the Java stored procedure wrapper
- Client-side Java code to call the PL/SQL wrapper

The Java stored procedure wraps the server-side Java code, which accomplishes the following:

- Wraps an instance method into a static method. Each method in the server-side Java code is wrapped by a static method. An instance method can be mapped in a single or multiple-instance fashion.
- Converts Java types into types that can be exposed to the PL/SQL call specification. For example, the Java type `byte[]` is converted into `oracle.sql.BLOB`.

The PL/SQL wrapper calls the Java stored procedure. The client-side Java code calls the PL/SQL wrapper through JDBC calls. The `-java` option requires that the class to be exposed is already loaded into the database.

The supported Java types are:

- JDBC supported types
- Java beans
- Arrays of supported types
- Serializable types

To publish a server-side class, use the `-dbjava` option, as follows:

```
-dbjava=server-sideClassName:client-sideClassName
```

The *`client-sideClassName`* setting must be specified. Otherwise, JPublisher will not generate client-side Java class. To publish `oracle.sqlj.checker.JdbcVersion`, use the following command:

```
% jpub -user=scott/tiger -dbjava=oracle.sqlj.checker.JdbcVersion:JdbcVersionClient
```

The command generates the following output:

```
oracle/sqlj/checker/JdbcVersionJPub.java
plsql_wrapper.sql
plsql_dropper.sql
SCOTT.JPUBTBL_VARCHAR2
SCOTT.JPUB_PLSQL_WRAPPER
Executing plsql_dropper.sql
Executing plsql_wrapper.sql
Loading JdbcVersionJPub.java
```

The command generates the `JdbcVersionJPub` Java stored procedure, the PL/SQL wrapper, and the client-side `JdbcVersionClient` class. `JdbcVersionJPub.java` and `plsql_wrapper.sql` are automatically loaded into the database. `JdbcVersionClient` has the following APIs:

```
public class JdbcVersionClient
{
  public JdbcVersionClient();
```

```
  public JdbcVersionClient(java.sql.Connection conn);
  public void setConnection(java.sql.Connection conn);
  public void setDataSource(javax.sql.DataSource ds);
  public String toString0();
  public java.math.BigDecimal getDriverMajorVersion();
  public java.math.BigDecimal getDriverMinorVersion();
  public String getDriverName();
  public String getDriverVersion();
  public String getJdbcLibraryName();
  public String getRecommendedRuntimeZip();
  public String getRuntimeVersion();
  public String getSqljLibraryName();
  public java.math.BigDecimal hasNewStatementCache();
  public java.math.BigDecimal hasOracleContextIsNew();
  public java.math.BigDecimal hasOracleSavepoint();
  public void main0(JpubtblVarchar2 arg0);
  public String to_string();
}
```

Compare `JdbcVersion` and `JdbcVersionClient`. It shows a limitation of JPublisher-generated code. The generated client-side APIs are not exactly the same as the original server-side APIs. To illustrate this limitation, the following is a list of several inconsistencies between `JdbcVersion` and `JdbcVersionClient`:

- The static methods are all mapped to instance methods, because a client-side method requires a JDBC connection to run.
- A client-side method always throws `java.sql.SQLException`, while exceptions thrown from the server-side class will be passed to the client wrapped with `SQLException`.
- The `toString()` method is renamed to `toString0()`. This is a limitation imposed by the stored procedure wrapper, where any method overwriting `java.lang.Object` methods has to be renamed to avoid conflicts.
- The parameter and return types may be different. Numeric types in the server-side are mapped to `java.math.BigDecimal`. Array types, such as `String[]`, are mapped to JDBC custom types. For example, the parameter of `main()` is mapped to `JpubtblVarchar2`, a subclass of `ORAData`, which the JPublisher command generates to represent an array of strings.
- The `main()` method in the server-side Java class will be renamed to `main0()`, due to the Java stored procedure limitation.

Compared to `-java`, the advantage of `-dbjava` is the support for more types and working with pre-10*g* database versions. However, the disadvantages are extra PL/SQL and Java stored procedure layers at run time and the increased possibility of change in the method signature in the client-side Java class.

Publishing Server-Side Java Classes to PL/SQL

JPublisher can generate PL/SQL wrappers for server-side Java classes. A Java class is mapped to a PL/SQL package. Each PL/SQL method corresponds to a Java method. This feature relieves the customer from writing the PL/SQL call specification and creating SQL types used in the call specification.

You can use the `-dbjava` option to generate the PL/SQL wrapper for a server-side Java class as follows:

`-dbjava=`*`server-sideJavaClass`*

Do *not* specify a name after *server-sideJavaClass*. Otherwise, JPublisher will map the server-side Java class to a client-side Java class.

> **See Also:** "Publishing Server-Side Java Classes Through PL/SQL Wrappers" on page 2-13

As an example, generate the PL/SQL wrapper for `oracle.sqlj.checker.JdbcVersion` using the following command:

```
% java -dbjava=oracle.sqlj.checker.JdbcVersion
```

The command generates the following output:

```
oracle/sqlj/checker/JdbcVersionJPub.java
plsql_wrapper.sql
plsql_dropper.sql
Executing plsql_dropper.sql
Executing plsql_wrapper.sql
Loading JdbcVersionJPub.java
```

The command generates and loads the Java stored procedure wrapper, `JdbcVersionJPub.java`, and also its PL/SQL wrapper, `plsql_wrapper.sql`, which declares the package `JPUB_PLSQL_WRAPPER`. The `JPUB_PLSQL_WRAPPER` package can be used to call the methods of `oracle.sqlj.checker.JdbcVersion`.

It often makes sense to specify `-plsqlfile` and `-plsqlpackage` with `-dbjava`. Consider the following command:

```
% java -dbjava=oracle.sqlj.checker.JdbcVersion -plsqlfile=jdbcversion.sql
-plsqlpackage=jdbcversion
```

The command generates the following output:

```
oracle/sqlj/checker/JdbcVersionJPub.java
jdbcversion.sql
jdbcversion_dropper.sql
Executing jdbcversion_dropper.sql
Executing jdbcversion.sql
Loading JdbcVersionJPub.java
```

The command generates `jdbcversion.sql`, which declares the `jdbcversion` PL/SQL package as the wrapper for `oracle.sqlj.checker.JdbcVersion`. The package is declared as follows:

```
CREATE OR REPLACE PACKAGE jdbcversion AS
  FUNCTION toString0 RETURN VARCHAR2;
  FUNCTION getDriverMajorVersion RETURN NUMBER;
  FUNCTION getDriverMinorVersion RETURN NUMBER;
  FUNCTION getDriverName RETURN VARCHAR2;
  FUNCTION getDriverVersion RETURN VARCHAR2;
  FUNCTION getJdbcLibraryName RETURN VARCHAR2;
  FUNCTION getRecommendedRuntimeZip RETURN VARCHAR2;
  FUNCTION getRuntimeVersion RETURN VARCHAR2;
  FUNCTION getSqljLibraryName RETURN VARCHAR2;
  FUNCTION hasNewStatementCache RETURN NUMBER;
  FUNCTION hasOracleContextIsNew RETURN NUMBER;
  FUNCTION hasOracleSavepoint RETURN NUMBER;
  PROCEDURE main0(arg0 JPUBTBL_VARCHAR2);
  FUNCTION to_string RETURN VARCHAR2;
END jdbcversion;
```

Note that the methods `toString()` and `main()` are renamed to `toString0()` and `main0()`, because of the Java stored procedure limitation.

You can run the PL/SQL stored procedures in the `jdbcversion` package as follows:

```
SQL> SELECT jdbcversion.toString0 FROM DUAL;

TOSTRING0
--------------------------------------------------------------------------------
Oracle JDBC driver version 10.2 (10.2.0.0.0)
SQLJ runtime: Oracle 9.2.0 for JDBC SERVER/JDK 1.2.x - Built on Oct 10, 2004
```

The `-dbjava` command publishes both static and instance methods. To publish the static method only, use the following setting:

```
-proxyopts=static
```

If the server-side class has a `public` empty constructor, then its instance methods can be published. Instance methods can be called in two ways, through a default single instance inside the server, or through individual instances. The following option determines the approach used to call instance methods inside the server:

```
-proxyopts=single|multiple
```

The default setting is:

```
-proxyopts=single
```

The preceding SQL statement calls the `toString0()` method using the single instance.

You can publish `oracle.sqlj.checker.JdbcVersion` using `-proxyopts=multiple`, as follows:

```
% jpub -user=scott/tiger -dbjava=oracle.sqlj.checker.JdbcVersion
-plsqlfile=jdbcversion.sql -plsqlpackage=jdbcversion -proxyopts=multiple
```

This command generates the `jdbcversion` PL/SQL package, with the following methods different from the previous example:

```
CREATE OR REPLACE PACKAGE jdbcversion AS
  FUNCTION toString0(handleJdbcVersion NUMBER) RETURN VARCHAR2;
  ...
  FUNCTION newJdbcVersion RETURN NUMBER;
END jdbcversion;
```

An extra method, `newJdbcVersion()`, is created this time. You can create an instance using this method and use the instance to call the `toString0()` method. Run the following script in SQL*Plus:

```
set serveroutput on
DECLARE
  text varchar2(1000);
  inst number;
BEGIN
  inst := jdbcversion.newJdbcVersion;
  text := jdbcversion.toString0(inst);
  dbms_output.put_line(text);
END;
/
```

This script returns:

```
Oracle JDBC driver version 10.2 (10.2.0.0.0)
SQLJ runtime: Oracle 9.2.0 for JDBC
SERVER/JDK 1.2.x - Built on Oct 10, 2004

PL/SQL procedure successfully completed.
```

The following parameter and return types are supported:

- JDBC supported types
- Java beans
- Arrays of supported types

Java beans are mapped to the generic JDBC `struct` class, `oracle.sql.STRUCT` at the Java stored procedure layer, and SQL object types and SQL table types at the PL/SQL layer. The following option determines how array parameters are handled:

```
-proxyopts=arrayin|arrayout|arrayinout|arrayall
```

The default setting is:

```
-proxyopts=arrayin
```

With `-proxyopts=arrayall`, a method containing array parameters is mapped to three PL/SQL methods. For example, consider the `foo(int[])` method. This method is mapped to the following methods:

```
PROCEDURE foo(n NUMBERTBL);
PRECEDURE foo_o(n IN NUMBER);
PROCEDURE foo_io(n IN OUT NUMBER);
```

The first method treats the array argument as an input, the second treats the array as a holder for an output value, and the third treats the array as a holder for both input and output values. With `-proxyopts=arrayin`, which is the default setting, the `foo(int[])` method is mapped to the first method. With `-proxyopts=arrayout`, the `foo(int[])` method is mapped to the second method. With `-proxyopts=arrayinout`, the `foo(int[])` method is mapped to the third method.

Consider a more complex example that uses two classes. The `Add` class uses `Total` and arrays in the methods. `Total` is a Java Bean and is therefore supported by server-side classes publishing. The two classes are defined as follows:

```
public class Add
{
  public static int[] add(int[] i, int[] j)
  {
    for (int k=0; k<i.length; k++)
      i[k] = i[k] + j[k];
    return i;
  }
  public int add(Total arg)
  {
    total = total + arg.getTotal();
    return total;
  }
  private int total;
}

public class Total
{
```

```
  public void setTotal(int total)
  {
    this.total = total;
  }
  public int getTotal()
  {
    return total;
  }
  private int total;
}
```

Load the two classes into the database, as follows:

```
% loadjava -u scott/tiger -r -v -f Add.java Total.java
```

Run JPublisher using the following command:

```
% jpub -user=scott/tiger -dbjava=Add  -proxyopts=arrayall
```

The command generates the following output:

```
AddJPub.java
plsql_wrapper.sql
plsql_dropper.sql
Executing plsql_dropper.sql
Executing plsql_wrapper.sql
Loading AddJPub.java
```

The generated PL/SQL wrapper, `plsql_wrapper.sql`, will have the following declaration:

```
CREATE OR REPLACE TYPE JPUBOBJ_Total AS OBJECT (total_ NUMBER);
CREATE OR REPLACE TYPE JPUBTBL_NUMBER AS TABLE OF NUMBER;
CREATE OR REPLACE PACKAGE JPUB_PLSQL_WRAPPER AS
  FUNCTION add(arg0  JPUBOBJ_Total) RETURN NUMBER;
  FUNCTION add_io(arg0  JPUBOBJ_Total) RETURN NUMBER;
  FUNCTION add(arg0  JPUBTBL_NUMBER,arg1  JPUBTBL_NUMBER) RETURN JPUBTBL_NUMBER;
  FUNCTION add_o(arg0 OUT  NUMBER,arg1 OUT  NUMBER) RETURN JPUBTBL_NUMBER;
  FUNCTION add_io(arg0 IN OUT  NUMBER,arg1 IN OUT NUMBER) RETURN JPUBTBL_NUMBER;
END JPUB_PLSQL_WRAPPER;
```

The following SQL script, when run in SQL*Plus, uses the generated PL/SQL wrapper:

```
SQL> set serveroutput on
SQL>
DECLARE
  totalx JPUBOBJ_Total;
  n NUMBER;
  n1 NUMBER;
  n2 NUMBER;
  add1 JPUBTBL_NUMBER;
  add2 JPUBTBL_NUMBER;
  add3 JPUBTBL_NUMBER;
BEGIN
  totalx := JPUBOBJ_Total(2004);
  n := JPUB_PLSQL_WRAPPER.add(totalx);
  n := JPUB_PLSQL_WRAPPER.add(totalx);
  DBMS_OUTPUT.PUT('total ');
  DBMS_OUTPUT.PUT_LINE(n);

  add1 := JPUBTBL_NUMBER(10, 20);
```

```
  add2 := JPUBTBL_NUMBER(100, 200);
  add3 := JPUB_PLSQL_WRAPPER.add(add1, add2);
  DBMS_OUTPUT.PUT('add ');
  DBMS_OUTPUT.PUT(add3(1));
  DBMS_OUTPUT.PUT(' ');
  DBMS_OUTPUT.PUT_LINE(add3(2));

  n1 := 99;
  n2 := 199;
  add3 := JPUB_PLSQL_WRAPPER.add_io(n1, n2);
  DBMS_OUTPUT.PUT('add_io ');
  DBMS_OUTPUT.PUT_LINE(n1);
END;
/
```

The script generates the following output:

```
total 4008
add 110 220
add_io 298
PL/SQL procedure successfully completed.
```

The `-dbjava` option requires the classes being published to be present in the database. You can use `-proxyclasses` instead, which requires the classes being published to be specified in the classpath. Compile `Add.java` and `Total.java`, and include `Add` and `Total` in the classpath. You can use the following command to publish `Add`, instead of the `-dbjava` option:

```
% jpub -proxyclasses=Add
```

The command generates the following output:

```
AddJPub.java
plsql_wrapper.sql
plsql_dropper.sql
Executing plsql_dropper.sql
Executing plsql_wrapper.sql
```

The `-proxyclasses` option loads the generated PL/SQL wrapper. However, it does not load the generated Java stored procedure, `AddJPub.java`, because this procedure requires the published classes to exist on the server. You need to load the Java stored procedure together with the published classes.

For example, on UNIX, you can load `Add.java`, `Total.java`, and `AddJPub.java` using the following command:

```
% loadjava -u scott/tiger -r -v -f Add.java Total.java AddJPub.java
```

Once `Add.java`, `Total.java`, and `AddJPub.java` are loaded, the PL/SQL wrapper is ready for use.

Mechanisms Used in Exposing Java to PL/SQL

JPublisher supports easy access to server-side Java classes by generating PL/SQL wrappers, otherwise known as PL/SQL call specifications. A PL/SQL wrapper is a PL/SQL package that can invoke methods of one or more given Java classes.

> **See Also:** *Oracle Database Java Developer's Guide* for information about PL/SQL wrappers

PL/SQL supports only static methods. Java classes with only static methods or classes for which you want to expose only static methods can be wrapped in a straightforward manner. However, for Java classes that have instance methods that you want to expose, an intermediate wrapper class is necessary to expose the instance methods as static methods for use by PL/SQL.

A wrapper class is also required if the Java class to be wrapped uses anything other than Java primitive types in its method calling sequences.

For instance methods in a class that is to be wrapped, JPublisher can use either or both of the following mechanisms in the wrapper class:

- Each wrapped class can be treated as a **singleton**, meaning that a single default instance is used. This instance is created the first time a method is called and is reused for each subsequent method call. Handles are not necessary and are not used. This mechanism is referred to as the **singleton mechanism** and is the default behavior when JPublisher provides wrapper classes for Web services client proxy classes.

  A `release`*`XXX`*`()` method is provided to remove the reference to the default instance and permit it to be garbage-collected.

- Instances of the wrapped class can be identified through **handles**, also known as ID numbers. JPublisher uses `long` numbers as handles and creates static methods in the wrapper class. The method signatures of these methods are modified to include the handle of the instance on which to invoke a method. This allows the PL/SQL wrapper to use the handles in accessing instances of the wrapped class. In this scenario, you must create an instance of each wrapped class to obtain a handle. Then you provide a handle for each subsequent instance method invocation. This mechanism is referred to as the **handle mechanism**.

  A `release`*`XXX`*`(long)` method is provided for releasing an individual instance according to the specified handle. A `releaseAll`*`XXX`*`()` method is provided for releasing all existing instances.

Publishing Server-Side Java Classes to Table Functions

The `-dbjava` option can generate table functions from the generated PL/SQL wrapper. Table functions are used if you want to expose data through database tables, rather than through stored function returns or stored procedure output values. A table function returns a database table.

> **See Also:** *Oracle Database PL/SQL User's Guide and Reference* for information about table functions.

For a table function to be generated for a given method, the following must be true:

- For wrapping instance methods, the singleton mechanism must be enabled. This is the default setting for `-dbjava` and `-proxyclasses`.
- The wrapped Web service method must correspond to a stored procedure with `OUT` arguments or to a stored function.

When used with the `-dbjava` or `-proxyclasses` option, the JPublisher `-proxyopts=tabfun` setting requests a table function created for each PL/SQL function in the generated PL/SQL wrapper. Consider the `Add` class example discussed earlier. Run the following command:

```
% jpub -user=scott/tiger -dbjava=Add  -proxyopts=arrayall,tabfun
```

The command generates the following output:

```
AddJPub.java
plsql_wrapper.sql
plsql_dropper.sql
Executing plsql_dropper.sql
Executing plsql_wrapper.sql
Loading AddJPub.java
```

This command generates the following extra table functions, in addition to the PL/SQL methods generated in the earlier example:

```
CREATE OR REPLACE PACKAGE JPUB_PLSQL_WRAPPER AS
   FUNCTION add(arg0 JPUBOBJ_Total) RETURN NUMBER;
   FUNCTION TO_TABLE_add(cur SYS_REFCURSOR) RETURN GRAPH_TAB_add_JPUBOBJ_Total
PIPELINED;
   FUNCTION add(arg0 JPUBTBL_NUMBER,arg1  JPUBTBL_NUMBER) RETURN JPUBTBL_NUMBER;
   FUNCTION TO_TABLE_add0(cur SYS_REFCURSOR) RETURN GRAPH_TAB_add_JPUBTBL_NUMBER
PIPELINED;
   FUNCTION add_o(arg0 OUT NUMBER, arg1 OUT NUMBER) RETURN JPUBTBL_NUMBER;
   FUNCTION TO_TABLE_add_o(cur SYS_REFCURSOR) RETURN
GRAPH_TAB_add_o_JPUBTBL_NUMBER PIPELINED;
   FUNCTION add_io(arg0 IN OUT NUMBER, arg1 IN OUT NUMBER) RETURN JPUBTBL_NUMBER;
   FUNCTION TO_TABLE_add_io(cur SYS_REFCURSOR) RETURN
GRAPH_TAB_add_io_JPUBTBL_NUMB PIPELINED;
END JPUB_PLSQL_WRAPPER;
/
```

The term, graph, is used with table functions. In this usage, a graph is a SQL object that defines the schema of the database table returned by a table function. There are three levels of functionality: a graph object, a table of graph objects, and a table function that returns the table of graph objects. The table of graph objects contains the input to a function and the output from that function.

As an example, consider the following declarations in `plsql_wrapper.sql`, which define the `GRAPH_add_io_JPUBTBL_NUMBER_J` graph object and the `GRAPH_TAB_add_io_JPUBTBL_NUMB` table of graph objects. These two types are generated for the `TO_TABLE_add_io` table function.

```
CREATE OR REPLACE TYPE GRAPH_add_io_JPUBTBL_NUMBER_J AS OBJECT(arg0 NUMBER, arg1
NUMBER, arg0_out NUMBER, arg1_out NUMBER, res JPUBTBL_NUMBER);
/

CREATE OR REPLACE TYPE GRAPH_TAB_add_io_JPUBTBL_NUMB AS TABLE OF
GRAPH_add_io_JPUBTBL_NUMBER_J;
/
```

Also note that a table function always takes a `REF CURSOR` as input. For the `TO_TABLE_add_io` table function, the `REF CURSOR` expects two arguments, `arg0` and `arg1`. The table function returns an instance of `GRAPH_TAB_add_io_JPUBTBL_NUMB`.

Run the following SQL script:

```
SQL> CREATE TABLE tabfun_input(arg0 NUMBER, arg1 NUMBER);
SQL> BEGIN
  INSERT INTO tabfun_input VALUES(97, 106);
  INSERT INTO tabfun_input VALUES(67, 3);
  INSERT INTO tabfun_input VALUES(19, 23);
  INSERT INTO tabfun_input VALUES(98, 271);
  INSERT INTO tabfun_input VALUES(83, 281);
```

```
END;
/
SQL>  SELECT * FROM TABLE(JPUB_PLSQL_WRAPPER.TO_TABLE_add_io(CURSOR(SELECT * FROM
tabfun_input)));
```

The query calls `TO_TABLE_add_io`, which shows the input and output of that table function.

```
    ARG0     ARG1 ARG0_OUT ARG1_OUT RES
-------- -------- -------- -------- -------------------------
      97      106      203      106 JPUBTBL_NUMBER(203)
      67        3       70        3 JPUBTBL_NUMBER(70)
      19       23       42       23 JPUBTBL_NUMBER(42)
      98      271      369      271 JPUBTBL_NUMBER(369)
      83      281      364      281 JPUBTBL_NUMBER(364)
```

Publishing Web Services Client into PL/SQL

JPublisher can publish a Web Service Description Language (WSDL) file into a PL/SQL package, to allow a database user to call a Web service from PL/SQL. This feature is called as **Web services call-out**. Given a WSDL file, JPublisher generates a Java-based Web services client proxy, and further generates PL/SQL wrapper for the client proxy. The client proxy is generated by the Oracle Database Web services assembler tool, which is started by JPublisher. Before starting the tool, the following have to be present in the database:

- The client proxy generated by JPublisher
- The PL/SQL wrapper generated by JPublisher
- The Java stored procedure wrapper generated by JPublisher
- The Java API for XML-based Remote Procedure Call (JAX-RPC) Web services client run time or Oracle Simple Object Access Protocol (SOAP) Web services client run time.

These components can be loaded automatically by JPublisher or manually by the user. At run time, a Web services call-out works as follows:

1. The user calls the PL/SQL wrapper, which in turn calls the Java stored procedure wrapper.
2. The Java stored procedure calls the client proxy.
3. The client proxy uses the Web services client run time to call the Web services.

The Java stored procedure wrapper is a required intermediate layer to publish instance methods of the client proxy class as static methods, because PL/SQL supports only static methods.

Web services call-out requires two JAR files, `dbwsa.jar` and `dbwsclient.jar`. These files are included in Database Web Services Callout Utility 10*g* release 2, which can be downloaded from:

http://www.oracle.com/technology/sample_code/tech/java/jsp/dbwebservices.html

Both `dbwsa.jar` and `dbwsclient.jar` should be copied to *ORACLE_HOME*`/sqlj/lib`. The `dbwsa.jar` file is required in the classpath when JPublisher publishes a WSDL file. On UNIX, the `jpub` command-line script includes the *ORACLE_HOME*`/sqlj/lib/dbwsa.jar`. Therefore, you do not have to include it

in the classpath. It is to be loaded into the database. It can be loaded into a user schema or into the SYS schema to be visible to other schemas. As an example, on UNIX, to load the .jar file into SYS, use the following loadjava command:

```
% loadjava -u sys/change_on_install -r -v -f -s -grant public
ORACLE_HOME/sqlj/lib/dbwsclient.jar
```

The dbwsclient.jar file includes client run time for both Oracle SOAP Web services and Oracle JAX- RPC Web services run time. This JAR file can be loaded into Oracle Database 10*g*. However, it cannot be loaded into Oracle9*i* Database.

For Oracle9*i* Database, only Oracle SOAP Web services client is supported. To load Oracle SOAP Web services client run time into a pre-9.2 Oracle Database, run the following command:

```
% loadjava -u sys/change_on_install -r -v -s -f -grant public \
                  ${J2EE_HOME}/lib/activation.jar \
                  ${J2EE_HOME}/lib/http_client.jar \
                  ${ORACLE_HOME}/lib/xmlparserv2.jar \
                  ${ORACLE_HOME}/soap/lib/soap.jar \
                  ${J2EE_HOME}/lib/mail.jar
```

The commands are in UNIX format. However, it gives an idea to Microsoft Windows users about the JAR files that are required for Oracle SOAP Web services client. The JAR files involved are distributed with Oracle9*i* Application Server releases.

To load Oracle SOAP Web services client into 9.2 or later releases of Oracle Database, run the following command:

```
% loadjava -u sys/change_on_install -r -v -s -f -grant public \
                  ${J2EE_HOME}/lib/jssl-1_2.jar \
                  ${ORACLE_HOME}/soap/lib/soap.jar \
                  ${ORACLE_HOME}/dms/lib/dms.jar \
                  ${J2EE_HOME}/lib/servlet.jar \
                  ${J2EE_HOME}/lib/ejb.jar \
                  ${J2EE_HOME}/lib/mail.jar
```

Web services call-outs require that JPublisher runs on JDK 1.4 or later. The following JPublisher options are related to Web services call-outs:

```
-proxywsdl=url
-httpproxy=host:port
-endpoint=url
-proxyopts=soap|jaxrpc|noload|tabfun. Default: -proxyopts=jaxrpc|tabfun.
-sysuser=user/password
```

where,

- The -proxywsdl option specifies the URL or path of a WSDL file, which describes the Web services being published.
- The -httpproxy option specifies the HTTP proxy that is used to access the WSDL file, if the file is outside a firewall.
- The -endpoint option redirects the client to the specified endpoint, rather than the endpoint specified in the WSDL file.

  See Also: "WSDL Document for Java and PL/SQL Wrapper Generation" on page 6-43 and "Web Services Endpoint" on page 6-44

- The -proxyopts=soap setting specifies that the PL/SQL wrapper will use the Oracle SOAP Web services client run time to call the Web services.

The `-proxyopts=jaxrpc` setting specifies that the PL/SQL wrapper will use Oracle JAX-RPC Web services client run time to call the Web services.

The `-proxyopts=tabfun` setting specifies that table functions be generated for applicable Web services operations.

- The `-sysuser` setting is recommended for `-proxywsdl`. It specifies a database user with `SYS` privileges. The `-sysuser` setting allows JPublisher to assign appropriate access privileges to run the generated PL/SQL wrappers. The `-sysuser` setting also allows JPublisher to load Web services client run time, if the run time is not present in the database.

For example, assume that a JAX-RPC Web service, called `HelloServiceEJB`, is deployed to the following endpoint:

```
http://localhost:8888/javacallout/javacallout
```

The WSDL document for this Web service is at the following location:

```
http://localhost:8888/javacallout/javacallout?WSDL
```

The Web service provides an operation called `getProperty` that takes a Java string specifying the name of a system property, and returns the value of that property. For example, `getProperty("os.name")` may return `SunOS`.

Based on the WSDL description of the Web service, JPublisher can direct the generation of a Web service client proxy, and generate Java and PL/SQL wrappers for the client proxy. Use the following command to perform these functions:

```
% jpub -user=scott/tiger -sysuser=sys/change_on_install
       -url=jdbc:oracle:thin:@localhost:1521:orcl
       -proxywsdl=http://localhost:8888/javacallout/javacallout?WSDL
       -package=javacallout -dir=genproxy
```

The command gives the following output:

```
genproxy/HelloServiceEJBJPub.java
genproxy/plsql_wrapper.sql
genproxy/plsql_dropper.sql
genproxy/plsql_grant.sql
genproxy/plsql_revoke.sql
Executing genproxy/plsql_wrapper.sql
Executing genproxy/plsql_grant.sql
Loading genproxy/plsql_proxy.jar
```

The `-proxyopts` setting directs the generation of the JAX-RPC client proxy and wrappers, and the use of a table function to wrap the Web service operation. The `-url` setting indicates the database, and the `-user` setting indicates the schema, where JPublisher loads the generated Java and PL/SQL wrappers. The `-sysuser` setting specifies the `SYS` account that has the privileges to grant permissions to run the wrapper script.

The `plsql_grant.sql` and `plsql_revoke.sql` scripts are generated by JPublisher. These scripts are used to create the PL/SQL wrapper in the database schema, grant permission to run it, revoke that permission, and drop the PL/SQL wrapper from the database schema.

The contents of the WSDL file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions name="HelloServiceEJB"
             targetNamespace="http://oracle.j2ee.ws/javacallout/Hello"
```

```
            xmlns:tns="http://oracle.j2ee.ws/javacallout/Hello"
            xmlns="http://schemas.xmlsoap.org/wsdl/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types/>
  <message name="HelloServiceInf_getProperty">
    <part name="String_1" type="xsd:string"/>
  </message>
  <message name="HelloServiceInf_getPropertyResponse">
    <part name="result" type="xsd:string"/>
  </message>
  <portType name="HelloServiceInf">
    <operation name="getProperty" parameterOrder="String_1">
      <input message="tns:HelloServiceInf_getProperty"/>
      <output message="tns:HelloServiceInf_getPropertyResponse"/>
    </operation>
  </portType>
  <binding name="HelloServiceInfBinding" type="tns:HelloServiceInf">
    <operation name="getProperty">
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded"
                   namespace="http://oracle.j2ee.ws/javacallout/Hello"/>
      </input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   use="encoded"
                   namespace="http://oracle.j2ee.ws/javacallout/Hello"/>
      </output>
      <soap:operation soapAction=""/>
    </operation>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
  </binding>
  <service name="HelloServiceEJB">
    <port name="HelloServiceInfPort" binding="tns:HelloServiceInfBinding">
      <soap:address location="/javacallout"/>
    </port>
  </service>
</definitions>
```

`HelloServiceInf` in the `<message>` element is the name of the service bean and determines the name of the interface that is generated and implemented by the generated JAX-RPC client proxy stub class. The `HelloServiceInf` interface has the following signature:

```
public interface HelloServiceInf extends java.rmi.Remote
{
  public String getProperty(String prop) throws java.rmi.RemoteException;
}
```

The method `getProperty()` corresponds to the `getProperty` operation specified in the WSDL document. It returns the value of a specified system property, *`prop`*. For example, specify the property `os.version` to return the operating system version.

The `plsql_wrapper.sql` file defines the `JPUB_PLSQL_WRAPPER` PL/SQL wrapper package. This package is created for calling the Web service from PL/SQL. It includes the definition of a table function from the Web service operation `getProperty`. The script in the `plsql_wrapper.sql` file is as follows:

```
CREATE OR REPLACE TYPE GRAPH_getProperty AS OBJECT(
  p0 VARCHAR2(32767),
```

```
  res VARCHAR2(32767)
);
/
CREATE OR REPLACE TYPE GRAPH_TAB_getProperty AS TABLE OF GRAPH_getProperty;
/
-- PL/SQL procedures that invoke webserviecs
CREATE OR REPLACE PACKAGE JPUB_PLSQL_WRAPPER AS
  FUNCTION getProperty(p0 VARCHAR2) RETURN VARCHAR2;
  FUNCTION TO_TABLE_getProperty(cur SYS_REFCURSOR) RETURN GRAPH_TAB_getProperty
PIPELINED;
END JPUB_PLSQL_WRAPPER;
/
```

Because the `-user` and `-sysuser` settings are specified in the JPublisher command line to publish this Web service, JPublisher will load the generated Java code and PL/SQL wrapper into the database. Once everything is loaded, you can use the PL/SQL wrapper to invoke the Web service.

The PL/SQL wrapper consists of two functions: `getProperty` and `TO_TABLE_getProperty`. The `getProperty` function directly wraps the `getProperty()` method in the generated client proxy class. For example, the following SQL*Plus command uses `getProperty` to determine the operating system where the Web service is running:

```
SQL> SELECT JPUB_PLSQL_WRAPPER.getProperty('os.name') FROM DUAL;
JPUB_PLSQL_WRAPPER.GETPROPERTY('OS.NAME')
----------------------------------------
SunOS
```

`TO_TABLE_getProperty` is a table function based on the `getProperty` function. It takes a `REF CURSOR` as input and returns a table. The schema of the table returned is defined by `GRAPH_getProperty`. In this example, `TO_TABLE_getProperty` is called with a `REF CURSOR` obtained from a one-column table of `VARCHAR2` data, where each data item is the name of a system property, such as `os.version`. `TO_TABLE_getProperty` returns a table in which each row contains an item from the input `REF CURSOR`, and the result of a `getProperty` call taking that item as input. The following code is a sample usage of `TO_TABLE_getProperty`:

```
SQL> -- Test Table Function
SQL> CREATE TABLE props (name VARCHAR2(50));
SQL> BEGIN
INSERT INTO props VALUES('os.version');
INSERT INTO props VALUES('java.version');
INSERT INTO props VALUES('file.separator');
INSERT INTO props VALUES('file.encoding.pkg');
INSERT INTO props VALUES('java.vm.info');
END;
/
SQL> SELECT * FROM
TABLE(JPUB_PLSQL_WRAPPER.TO_TABLE_getProperty(CURSOR(SELECT * FROM props)));
P0 RES
------------------------------
os.version 5.8
java.version 1.4.1_03
file.separator /
file.encoding.pkg sun.io
java.vm.info mixed mode
```

This example creates a one-column table of `VARCHAR2`, populates it with system property names, and uses `TO_TABLE_getProperty` to find out the values of those

system properties. In this example, you can see that the operating system is Sun Microsystems Solaris 5.8.

3

Data Type and Java-to-Java Type Mappings

This chapter discusses the JPublisher support for data type mapping, including a section on JPublisher styles and style files for Java-to-Java type mappings. These style files are primarily used to provide Web services support. The chapter contains the following sections:

- JPublisher Data Type Mappings
- Support for PL/SQL Data Types
- JPublisher Styles and Style Files

JPublisher Data Type Mappings

This section covers the JPublisher functionality for mapping from SQL and PL/SQL to Java in the following topics:

- Overview of JPublisher Data Type Mappings
- SQL and PL/SQL Mappings to Oracle and JDBC Types
- JPublisher User Type Map and Default Type Map
- JPublisher Logical Progression for Data Type Mappings
- Object Attribute Types
- REF CURSOR Types and Result Sets Mapping
- Connection in JDBC Mapping

See Also: "Support for PL/SQL Data Types" on page 3-10

Overview of JPublisher Data Type Mappings

When you use the `-builtintypes`, `-lobtypes`, `-numbertypes`, and `-usertypes` type mapping options, you can specify one of the following settings for data type mappings:

- `oracle`
- `jdbc`
- `objectjdbc`
- `bigdecimal`

Note: The `objectjdbc` and `bigdecimal` settings are for the `-numbertypes` options only.

These mappings affect the argument and result types that JPublisher uses in the methods it generates.

The class that JPublisher generates for an object type has the `get`*`XXX`*`()` and `set`*`XXX`*`()` accessor methods for the object attributes. The class that JPublisher generates for a `VARRAY` or nested table type has the `get`*`XXX`*`()` and `set`*`XXX`*`()` methods, which access the elements of the array or nested table. When generation of wrapper methods is enabled, the class that JPublisher generates for an object type or PL/SQL package has wrapper methods. These wrapper methods invoke server methods, or stored procedures, of the object type or package. The mapping options control the argument and result types that these methods use.

The Java Database Connectivity (JDBC) and Object JDBC mappings use familiar Java types that can be manipulated using standard Java operations. The Oracle mapping is the most efficient mapping. The `oracle.sql` types match the Oracle internal data types as closely as possible so that little or no data conversion is required between the Java and SQL formats. You do not lose any information and have greater flexibility in how you process and unpack the data. If you are manipulating data or moving data within the database, then the Oracle mappings for standard SQL types are the most convenient representations. For example, performing `SELECT` and `INSERT` operations from one existing table to another. When data format conversion is necessary, you can use methods in the `oracle.sql.*` classes to convert to Java native types.

SQL and PL/SQL Mappings to Oracle and JDBC Types

Table 3–1 lists the mappings from SQL and PL/SQL data types to Java types. You can use all the supported data types listed in this table as argument or result types for PL/SQL methods. You can also use a subset of the data types as object attribute types.

> **See Also:** "Object Attribute Types" on page 3-7

The SQL and PL/SQL Data Type column contains all possible data types.

The Oracle Mapping column lists the corresponding Java types that JPublisher uses when all the type mapping options are set to `oracle`. These types are found in the `oracle.sql` package provided by Oracle and are designed to minimize the overhead incurred when converting Oracle data types to Java types.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for more information on the `oracle.sql` package

The JDBC Mapping column lists the corresponding Java types that JPublisher uses when all the type mapping options are set to `jdbc`. For standard SQL data types, JPublisher uses Java types specified in the JDBC specification. For SQL data types that are Oracle extensions, JPublisher uses the `oracle.sql.*` types. When you set the `-numbertypes` option to `objectjdbc`, the corresponding types are the same as in the JDBC Mapping column, except that primitive Java types, such as `int`, are replaced with their object counterparts, such as `java.lang.Integer`.

Note: Type correspondences explicitly defined in the JPublisher type map, such as PL/SQL `BOOLEAN` to SQL `NUMBER` to Java `boolean`, are not affected by the mapping option settings.

A few data types are not directly supported by JPublisher, in particular those types that pertain only to PL/SQL. You can overcome these limitations by providing equivalent SQL and Java types, as well as PL/SQL conversion functions between

PL/SQL and SQL representations. The annotations and subsequent sections explain these conversions further.

Table 3–1 SQL and PL/SQL Data Type to Oracle and JDBC Mapping Classes

SQL and PL/SQL Data Type	Oracle Mapping	JDBC Mapping
CHAR, CHARACTER, LONG, STRING, VARCHAR, VARCHAR2	oracle.sql.CHAR	java.lang.String
NCHAR, NVARCHAR2	oracle.sql.NCHAR (note 1)	oracle.sql.NString (note 1)
NCLOB	oracle.sql.NCLOB (note 1)	oracle.sql.NCLOB (note 1)
RAW, LONG RAW	oracle.sql.RAW	byte[]
BINARY_INTEGER, NATURAL, NATURALN, PLS_INTEGER, POSITIVE, POSITIVEN, SIGNTYPE, INT, INTEGER	oracle.sql.NUMBER	int
DEC, DECIMAL, NUMBER, NUMERIC	oracle.sql.NUMBER	java.math.BigDecimal
DOUBLE PRECISION, FLOAT	oracle.sql.NUMBER	double
SMALLINT	oracle.sql.NUMBER	int
REAL	oracle.sql.NUMBER	float
DATE	oracle.sql.DATE	java.sql.Timestamp
TIMESTAMP	oracle.sql.TIMESTAMP	java.sql.Timestamp
TIMESTAMP WITH TZ	oracle.sql.TIMESTAMPTZ	
TIMESTAMP WITH LOCAL TZ	oracle.sql.TIMESTAMPLTZ	
INTERVAL YEAR TO MONTH	String (note 2)	String (note 2)
INTERVAL DAY TO SECOND		
ROWID, UROWID	oracle.sql.ROWID	oracle.sql.ROWID
BOOLEAN	boolean (note 3)	boolean (note 3)
CLOB	oracle.sql.CLOB	java.sql.Clob
BLOB	oracle.sql.BLOB	java.sql.Blob
BFILE	oracle.sql.BFILE	oracle.sql.BFILE
Object types	Generated class	Generated class
SQLJ object types	Java class defined at type creation	Java class defined at type creation
OPAQUE types	Generated or predefined class (note 4)	Generated or predefined class (note 4)
RECORD types	Through mapping to SQL object type (note 5)	Through mapping to SQL object type (note 5)
Nested table, VARRAY	Generated class implemented using oracle.sql.ARRAY	java.sql.Array
Reference to object type	Generated class implemented using oracle.sql.REF	java.sql.Ref
REF CURSOR	java.sql.ResultSet	java.sql.ResultSet

Table 3–1 (Cont.) SQL and PL/SQL Data Type to Oracle and JDBC Mapping Classes

SQL and PL/SQL Data Type	Oracle Mapping	JDBC Mapping
Index-by tables	Through mapping to SQL collection (note 6)	Through mapping to SQL collection (note 6)
Scalar (numeric or character) Index-by tables	Through mapping to Java array (note 7)	Through mapping to Java array (note 7)
User-defined subtypes	Same as for base type	Same as for base type

Data Type Mapping Notes The following notes correspond to the entries in the preceding table:

1. The Java classes `oracle.sql.NCHAR`, `oracle.sql.NCLOB`, and `oracle.sql.NString` are not part of JDBC but are distributed with the JPublisher run time. JPublisher uses these classes to represent the `NCHAR` form of use of the corresponding classes, `oracle.sql.CHAR`, `oracle.sql.CLOB`, and `java.lang.String`.
2. Mappings of SQL `INTERVAL` types to the Java `String` type are defined in the JPublisher default type map. Functions from the `SYS.SQLJUTL` package are used for the conversions.

   See Also: "JPublisher User Type Map and Default Type Map" on page 3-5

3. Mapping of PL/SQL `BOOLEAN` to SQL `NUMBER` and Java `boolean` is defined in the default JPublisher type map. This process uses conversion functions from the `SYS.SQLJUTL` package.
4. Mapping of the `SYS.XMLTYPE` SQL `OPAQUE` type to the `oracle.xdb.XMLType` Java class is defined in the default JPublisher type map. For other `OPAQUE` types, the vendor typically provides a corresponding Java class. In this case, you must specify a JPublisher type map entry that defines the correspondence between the SQL `OPAQUE` type and the corresponding Java wrapper class. If JPublisher encounters an `OPAQUE` type that does not have a type map entry, then it generates a Java wrapper class for that `OPAQUE` type.

   See Also: "Type Mapping Support for OPAQUE Types" on page 3-11

5. To support a PL/SQL `RECORD` type, JPublisher maps the `RECORD` type to a SQL object type and then to a Java type corresponding to the SQL object type. JPublisher generates two SQL scripts. One script is to create the SQL object type and to create a PL/SQL package containing the conversion functions between the SQL type and the `RECORD` type. The other script is used to drop the SQL type and the PL/SQL package created by the first script.

   See Also: "Type Mapping Support for PL/SQL RECORD and Index-By Table Types" on page 3-19

6. To support a PL/SQL index-by table type, JPublisher first maps the index-by table type into a SQL collection type and then maps it into a Java class corresponding to that SQL collection type. JPublisher generates two SQL scripts. One to create the SQL collection type and to create a PL/SQL package containing conversion functions between the SQL collection type and the index-by table type. The other to drop the collection type and the PL/SQL package created by the first script.

See Also: "Type Mapping Support for PL/SQL RECORD and Index-By Table Types" on page 3-19

7. If you use the JDBC driver to call PL/SQL stored procedures or object methods, then you have direct support for scalar index-by tables, also known as PL/SQL `TABLE` types. In this case, you need to use a type map entry for JPublisher that specifies the PL/SQL scalar index-by table type and a corresponding Java array type. JPublisher can then automatically publish PL/SQL or object method signatures that use this scalar index-by type.

   See Also: "Type Mapping Support for Scalar Index-by Tables" on page 3-13

JPublisher User Type Map and Default Type Map

JPublisher has a user type map, which is controlled by the `-typemap` and `-addtypemap` options and starts out empty. It also has a default type map, which is controlled by the `-defaulttypemap` and `-adddefaulttypemap` options and starts with entries such as the following:

```
jpub.defaulttypemap=SYS.XMLTYPE:oracle.xdb.XMLType
jpub.adddefaulttypemap=BOOLEAN:boolean:INTEGER:
SYS.SQLJUTL.INT2BOOL:SYS.SQLJUTL.BOOL2INT
jpub.adddefaulttypemap=INTERVAL DAY TO SECOND:String:CHAR:
SYS.SQLJUTL.CHAR2IDS:SYS.SQLJUTL.IDS2CHAR
jpub.adddefaulttypemap=INTERVAL YEAR TO MONTH:String:CHAR:
SYS.SQLJUTL.CHAR2IYM:SYS.SQLJUTL.IYM2CHAR
```

These commands, which include some wraparound lines, indicate mappings between PL/SQL types, Java types, and SQL types. Where applicable, they also specify conversion functions to convert between PL/SQL types and SQL types.

See Also: "Type Map Options" on page 6-25

JPublisher checks the default type map first. If you attempt in the user type map to redefine a mapping that is in the default type map, JPublisher generates a warning message and ignores the redefinition. Similarly, attempts to add mappings through `-adddefaulttypemap` or `-addtypemap` settings that conflict with previous mappings are ignored and generate warnings.

There are typically two scenarios for using the type maps:

- Specify type mappings for PL/SQL data types that are unsupported by JDBC.
- Avoid regenerating a Java class to map to a user-defined type. For example, assume you have a user-defined SQL object type, `STUDENT`, and have already generated a `Student` class to map to it. If you specify the `STUDENT:Student` mapping in the user type map, then JPublisher finds the `Student` class and uses it for mapping without regenerating it.

  See Also: "Example: Using the Type Map to Avoid Regeneration"

To use custom mappings, it is recommended that you clear the default type map, as follows:

```
-defaulttypemap=
```

Then use the `-addtypemap` option to put any required mappings into the user type map.

The predefined default type map defines a correspondence between the SYS.XMLTYPE SQL OPAQUE type and the oracle.xdb.XMLType Java wrapper class. In addition, it maps the PL/SQL BOOLEAN type to the Java boolean type and the SQL INTEGER type through two conversion functions defined in the SYS.SQLJUTL package. Also, the default type map provides mappings between the SQL INTERVAL type and the Java String type.

However, you may prefer mapping the PL/SQL BOOLEAN type to the Java object type Boolean to capture the SQL NULL values in addition to the true and false values. You can accomplish this by resetting the default type map, as shown by the following:

```
-defaulttypemap=BOOLEAN:Boolean:INTEGER:SYS.SQLJUTL.INT2BOOL:SYS.SQLJUTL.BOOL2INT
```

This changes the designated Java type from boolean to Boolean, as well as eliminating any other existing default type map entries. The rest of the conversion remains valid.

Example: Using the Type Map to Avoid Regeneration The following example uses the JPublisher type map to avoid the mapping of regenerated Java classes. Assume the following type declarations, noting that the CITY type is an attribute of the TRIP type:

```
SQL> CREATE TYPE city AS OBJECT (name VARCHAR2(20), state VARCHAR2(10));
/
SQL> CREATE OR REPLACE TYPE trip AS OBJECT (leave DATE, place city);
/
```

Now assume that you invoke JPublisher as follows:

```
% jpub -u scott/tiger -s TRIP:Trip
```

The JPublisher output is:

```
SCOTT.TRIP
SCOTT.CITY
```

Only TRIP is specified for processing. However, the command produces the source files City.java, CityRef.java, Trip.java, and TripRef.java, because CITY is an attribute.

If you want to regenerate the classes for TRIP without regenerating the classes for CITY, then you can rerun JPublisher as follows:

```
% jpub -u scott/tiger -addtypemap=CITY:City -s TRIP:Trip
SCOTT.TRIP
```

As you can see from the output line, the CITY type is not reprocessed and, therefore, the City.java and CityRef.java files are not regenerated. This is because of the addition of the CITY:City relationship to the type map, which informs JPublisher that the existing City class is to be used for mapping.

JPublisher Logical Progression for Data Type Mappings

To map a given SQL or PL/SQL type to Java, JPublisher uses the following logical progression:

1. Checks the type maps to see if the mapping is already specified.
2. Checks the predefined Java mappings for SQL and PL/SQL types.
3. Checks whether the data type to be mapped is a PL/SQL RECORD type or an index-by table type. If it is a PL/SQL RECORD type, JPublisher generates a

corresponding SQL object type that it can then map to Java. If it is an index-by table type, JPublisher generates a corresponding SQL collection type that it can then map to Java.

4. If none of steps 1 through 3 apply, then the data type must be a user-defined type. JPublisher generates an `ORAData` or `SQLData` class to map it according to the JPublisher option settings.

Object Attribute Types

You can use a subset of the SQL data types in Table 3–1 as object attribute types. The types that can be used are listed here:

- `CHAR`, `VARCHAR`, `VARCHAR2`, `CHARACTER`
- `NCHAR`, `NVARCHAR2`
- `DATE`
- `DECIMAL`, `DEC`, `NUMBER`, `NUMERIC`
- `DOUBLE PRECISION`, `FLOAT`
- `INTEGER`, `SMALLINT`, `INT`
- `REAL`
- `RAW`, `LONG RAW`
- `CLOB`
- `BLOB`
- `BFILE`
- `NCLOB`
- Object type, `OPAQUE` type, SQLJ object type
- Nested table, `VARRAY` type
- Object reference type

JPublisher supports the following TIMESTAMP types as object attributes:

- `TIMESTAMP`
- `TIMESTAMP WITH TIMEZONE`
- `TIMESTAMP WITH LOCAL TIMEZONE`

> **Note:** The Oracle JDBC implementation does not support the TIMESTAMP types.

REF CURSOR Types and Result Sets Mapping

If a PL/SQL stored procedure or function or a SQL query returns a `REF CURSOR`, then JPublisher generates a method, by default, to map the `REF CURSOR` to `java.sql.ResultSet`.

In addition, for a SQL query, but not for a `REF CURSOR` returned by a stored procedure or function, JPublisher generates a method to map the `REF CURSOR` to an array of rows. In this array, each row is represented by a JavaBean instance.

In addition, with a setting of `-style=webservices-common`, if the following classes are available in the classpath, then JPublisher generates methods to map the `REF CURSOR` to the following types:

- `javax.xml.transform.Source`
- `oracle.jdbc.rowset.OracleWebRowSet`
- `org.w3c.dom.Document`

Notes:

- The dependency of having the class in the classpath in order to generate the mapping is specified by a `CONDITION` statement in the style file. The `CONDITION` statement lists required classes.
- The `webservices9` and `webservices10` style files include `webservices-common`, but override these mappings. Therefore, JPublisher will *not* produce these mappings with a setting of `-style=webservices9` or `-style=webservices10`.

If required, you need to perform the following actions to ensure that JPublisher can find the classes:

1. Ensure that the libraries `translator.jar`, `runtime12.jar`, and `classes12.jar` or `ojdbc14.jar` are in the classpath. These files contain JPublisher and SQLJ translator classes, SQLJ run time classes, and JDBC classes, respectively.
2. Use Java Development Kit (JDK) 1.4, for mapping to `Source`. This class is not defined in earlier JDK versions.
3. Add *`ORACLE_HOME`*`/jdbc/lib/rowset-jsr114.jar` to the classpath, for mapping to `OracleWebRowSet`.
4. Add *`ORACLE_HOME`*`/lib/xmlparsev2.jar` to the classpath, for mapping to `Document`.

Consider the following PL/SQL stored procedure:

```
TYPE curtype1 IS REF CURSOR RETURN emp%rowtype;
FUNCTION get1 RETURN curtype1;
```

If the `OracleWebRowSet` class is found in the classpath during publishing, but `Document` and `Source` are not, then JPublisher generates the following methods for the `get1` function:

```
public oracle.jdbc.rowset.OracleWebRowSet get1WebRowSet()
                                throws java.sql.SQLException;
public java.sql.ResultSet get1() throws java.sql.SQLException;
```

The names of methods returning `Document` and `Source` would be `get1XMLDocument()` and `get1XMLSource()`, respectively.

Disabling Mapping to Source, OracleWebRowSet, or Document

There is currently no JPublisher option to explicitly enable or disable mapping to `Source`, `OracleWebRowSet`, or `Document`. The only condition in the `webservices-common` style file is whether the classes exist in the classpath. However, you can copy and edit your own style file if you want more control over how JPublisher maps `REF CURSOR`. The following code is an excerpt from the

webservices-common file that has been copied and edited as an example. Descriptions of the edits follow the code.

```
BEGIN_TRANSFORMATION
MAPPING
SOURCETYPE java.sql.ResultSet
TARGETTYPE java.sql.ResultSet
RETURN
%2 = %1;
END_RETURN;
END_MAPPING

MAPPING
#CONDITION oracle.jdbc.rowset.OracleWebRowSet
SOURCETYPE java.sql.ResultSet
TARGETTYPE oracle.jdbc.rowset.OracleWebRowSet
TARGETSUFFIX WebRowSet
RETURN
%2 = null;
if (%1!=null)
{
  %2 = new oracle.jdbc.rowset.OracleWebRowSet();
  %2.populate(%1);
}
END_RETURN
END_MAPPING

#MAPPING
#CONDITION org.w3c.dom.Document oracle.xml.sql.query.OracleXMLQuery
#SOURCETYPE java.sql.ResultSet
#TARGETTYPE org.w3c.dom.Document
#TARGETSUFFIX XMLDocument
#RETURN
#%2 = null;
#if (%1!=null)
#  %2= (new oracle.xml.sql.query.OracleXMLQuery
#                                  (_getConnection(), %1)).getXMLDOM();
#END_RETURN
#END_MAPPING

MAPPING
CONDITION org.w3c.dom.Document oracle.xml.sql.query.OracleXMLQuery
          javax.xml.transform.Source javax.xml.transform.dom.DOMSource
SOURCETYPE java.sql.ResultSet
TARGETTYPE javax.xml.transform.Source
TARGETSUFFIX XMLSource
RETURN
%2 = null;
if (%1!=null)
  %2= new javax.xml.transform.dom.DOMSource
      ((new oracle.xml.sql.query.OracleXMLQuery
       (new oracle.xml.sql.dataset.OracleXMLDataSetExtJdbc(_getConnection(),
       (oracle.jdbc.OracleResultSet) %1))).getXMLDOM());
END_RETURN
END_MAPPING
END_TRANSFORMATION
```

Assume that you copy this file into myrefcursormaps.properties. There are four MAPPING sections intended for mapping REF CURSOR to ResultSet, OracleWebRowSet, Document, and Source according to the SOURCETYPE and

TARGETTYPE entries. For this example, lines are commented out using the "#" character to accomplish the following:

- The CONDITION statement is commented out for the OracleWebRowSet mapping. Because of this, JPublisher will generate a method for this mapping regardless of whether OracleWebRowSet is in the classpath.
- The entire MAPPING section is commented out for the Document mapping. JPublisher will not generate a method for this mapping.

Run JPublisher with the following options to use your custom mappings:

```
% jpub -u scott/tiger -style=myrefcursormaps -s MYTYPE:MyType
```

Connection in JDBC Mapping

With the -usertypes=jdbc setting, JPublisher generates SQLData for a SQL object type. The underlying JDBC connection for a SQLData instance is not automatically set by the JDBC driver. Therefore, before accessing attributes in a SQLData instance, you must set a JDBC connection using the setConnectionContext() method.

Consider Address is a SQLData class generated by JPublisher with -usertypes=jdbc. The following code segment accesses the attribute of an Address instance. Note that the setConnectionContext call explicitly initializes the underlying JDBC connection.

```
...
ResultSet rset = stmt.executeQuery();
Address address = (Address) rset.getObject(1);
address.setConnectionContext(new sqlj.runtime.ref.DefaultContext(connection));
String addr = address.getAddress();
...
```

Note: Other -usertypes settings do not require setting the connection, as described in the preceding code example.

On the other hand, for ORAData types that JPublisher generates with the -usertypes=oracle setting or by default, connection initialization is not required. The underlying JDBC connection for ORAData is already assigned at the time it is read from ResultSet.

Support for PL/SQL Data Types

There are three scenarios if JPublisher encounters a PL/SQL stored procedure or function, including method of a SQL object type, which uses a PL/SQL type that is unsupported by JDBC:

- If you specify a mapping for the PL/SQL type in the default type map or user type map, then JPublisher uses that mapping.

  See Also: "JPublisher User Type Map and Default Type Map" on page 3-5

- If there is no mapping in the type maps, and the PL/SQL type is a RECORD type or an index-by table type, then JPublisher generates a corresponding SQL type that JDBC supports. For a PL/SQL RECORD type, JPublisher generates a SQL object

type to bridge between the RECORD type and Java. For an index-by table type, JPublisher generates a SQL collection type for the bridge.

- If neither of the first two scenarios applies, then JPublisher issues a warning message and uses <unsupported type> in the generated code to represent the unsupported PL/SQL type.

The following sections discuss further details of JPublisher type mapping features for PL/SQL types unsupported by JDBC:

Type Mapping Support for OPAQUE Types

This section describes JPublisher type mapping support for SQL OPAQUE types in general and the SYS.XMLTYPE SQL OPAQUE type in particular. It covers the following topics:

> **Note:** If you want JPublisher to generate wrapper classes for SQL OPAQUE types, then you must use an Oracle9*i* Database release 2 (9.2) or later installation and JDBC driver.

Support for OPAQUE Types

The Oracle JDBC and SQLJ implementations support SQL OPAQUE types published as Java classes implementing the oracle.sql.ORAData interface. Such classes must contain the following public, static fields and methods:

```
public static String _SQL_NAME = "SQL_name_of_OPAQUE_type";
public static int _SQL_TYPECODE = OracleTypes.OPAQUE;
public static ORADataFactory getORADataFactory() { ... }
```

If you have a Java wrapper class to map to a SQL OPAQUE type, and the class meets this requirement, then you can specify the mapping through the JPublisher user type map. Use the -addtypemap option with the following syntax to append the mapping to the user type map:

```
-addtypemap=sql_opaque_type:java_wrapper_class
```

In Oracle Database 10*g*, the SYS.XMLTYPE SQL OPAQUE type is mapped to the oracle.xdb.XMLType Java class through the JPublisher default type map. You could accomplish the same thing explicitly through the user type map, as follows:

```
-addtypemap=SYS.XMLTYPE:oracle.xdb.XMLType
```

Whenever JPublisher encounters a SQL OPAQUE type for which no type correspondence has been provided, it publishes a Java wrapper class. Consider the following SQL type defined in the SCOTT schema:

```
CREATE TYPE X_TYP AS OBJECT (xml SYS.XMLTYPE);
```

The following command publishes `X_TYP` as a Java class `XTyp`:

```
% jpub -u scott/tiger -s X_TYP:XTyp
```

By default, the `xml` attribute is published using `oracle.xdb.XMLType`, which is the predefined type mapping for `SYS.XMLTYPE`. If you clear the JPublisher default type map, then a wrapper class, `Xmltype`, will automatically be generated for the `SYS.XMLTYPE` attribute. You can verify this by invoking JPublisher as follows:

```
% jpub -u scott/tiger -s X_TYP:XTyp -defaulttypemap=
```

The `-defaulttypemap` option is for setting the JPublisher default type map. Giving it no value, as in the preceding example, clears it.

> **See Also:** "JPublisher User Type Map and Default Type Map" on page 3-5 and "Type Map Options" on page 6-25

Support for XMLTYPE

In Oracle Database 10*g*, the `SYS.XMLTYPE` SQL `OPAQUE` type is supported with the `oracle.xdb.XMLType` Java class located in *`ORACLE_HOME`*`/lib/xsu12.jar`. This class is the default mapping, but it requires the Oracle Database 10*g* JDBC Oracle Call Interface (OCI) driver. It is currently not supported by the JDBC Thin driver.

The SQLJ run time provides the Java class `oracle.sql.SimpleXMLType` as an alternative mapping for `SYS.XMLTYPE`. This works on both the OCI driver and the Thin driver. With the following setting, JPublisher maps `SYS.XMLTYPE` to `oracle.sql.SimpleXMLType`:

```
-adddefaulttypemap=SYS.XMLTYPE:oracle.sql.SimpleXMLType
```

`SimpleXMLType`, defined in `runtime12.jar`, can read an `XMLTYPE` instance as a `java.lang.String` instance, or create an `XMLTYPE` instance from a `String` instance.

For Java-to-Java type transformations, which is often necessary for Web services, the `webservices-common.properties` style file specifies the preceding mapping as well as the Java-to-Java mapping of `SimpleXMLType` to `java.lang.String`. With a setting of `-style=webservices-common`, JPublisher maps `SYS.XMLTYPE` to `SimpleXMLType` in the generated base Java class and to `String` in the user subclass.

> **See Also:** "JPublisher-Generated Subclasses for Java-to-Java Type Transformations" on page 4-16 and "JPublisher Styles and Style Files" on page 3-23

The `webservices9.properties` and `webservices10.properties` style files include the `webservices-common.properties` file. However, these files override the Java-to-Java mapping from `SimpleXMLType` to `String`. The `webservices9.properties` file maps `SimpleXMLType` to `org.w3c.dom.DocumentFragment` for the user subclass and the `webservices10.properties` file maps it to `javax.xml.transform.Source`.

Consider a setting of `-style=webservices9` as an example. The user subclass converts from `SimpleXMLType` to `DocumentFragment` or from `DocumentFragment` to `SimpleXMLType`, so that a SQL or PL/SQL method using `SYS.XMLTYPE` can be exposed as a Java method by using `org.w3c.dom.DocumentFragment`. The following example contains the JPublisher

command line and portions of the PL/SQL procedure, the Java interface, the base Java class, and the user subclass.

The JPublisher command on the command line is:

```
% jpub -u scott/tiger -sql=xtest:XTestBase:XTestUser#XTest -style=webservices9
```

The SQL definitions are:

```
PROCEDUR setXMLMessage(x XMLTYPE, y NUMBER);
FUNCTION getXMLMessage(id NUMBER) RETURN XMLTYPE;
```

The definitions in the `XTest` interface are:

```
public org.w3c.dom.DocumentFragment getxmlmessage(java.math.BigDecimal id)
public void setxmlmessage(org.w3c.dom.DocumentFragment x,
                          java.math.BigDecimal y)
```

The definitions in `XTestBase.java` are:

```
public oracle.sql.SimpleXMLType _getxmlmessage (java.math.BigDecimal id)
public void _setxmlmessage (oracle.sql.SimpleXMLType x, java.math.BigDecimal y)
```

The definitions in `XTestUser.java` are:

```
public org.w3c.dom.DocumentFragment getxmlmessage(java.math.BigDecimal id)
public void setxmlmessage(org.w3c.dom.DocumentFragment x,
                          java.math.BigDecimal y)
```

Type Mapping Support for Scalar Index-by Tables

The term **scalar PL/SQL index-by table** refers to a PL/SQL index-by table with elements of `VARCHAR` and numerical types. Starting 10*g* Release 1 (10.2), JPublisher can map a simple PL/SQL index-by table into a Java array, as an alternative to mapping PL/SQL index-by tables into custom JDBC types. The new option `plsqlindextable` specifies how a simple PL/SQL index-by table is mapped.

```
-plsqlindextable=custom|array|int
```

If `-plsqlindextable=custom` is set, all indexby tables are mapped to custom JDBC types, such as `SQLData`, `CustomDatum`, or `ORAData`. If `-plsqlindextable=array` or `-plsqlindextable=`*`int`* is set, a simple index-by table will be mapped to a Java array. With `-plsqlindextable=`*`int`*, the *`int`* value specifies the array capacity, which is 32768 by default. The default setting for this option is `custom`.

Consider the following PL/SQL package:

```
CREATE OR REPLACE PACKAGE indexbytable_package AS
  TYPE index_tbl1 IS TABLE OF VARCHAR2(111) INDEX BY binary_integer;
  TYPE index_tbl2 IS TABLE OF NUMBER INDEX BY binary_integer;
  TYPE varray_tbl3 IS VARRAY(100) OF VARCHAR2(20);
  TYPE nested_tbl4 IS TABLE OF VARCHAR2(20);
  FUNCTION echo_index_tbl1(a index_tbl1) RETURN index_tbl1;
  FUNCTION echo_index_tbl2(a index_tbl2) RETURN index_tbl2;
  FUNCTION echo_varray_tbl3(a varray_tbl3) RETURN varray_tbl3;
  FUNCTION echo_nested_tbl4(a nested_tbl4) RETURN nested_tbl4;
END;
/
```

Run the following command:

```
% jpub -u scott/tiger
-sql=indexbytable_package:IndexbyTablePackage#IndexbyTableIntf -plsqlindextable=32
```

The `-plsqlindextable=32` setting specifies that simple index-by tables are mapped to Java arrays, with a capacity of 32. The following interface is generated in `IndexbyTableIntf.java`:

```
public interface IndexbyTableIntf
{
 public String[] echoIndexTbl1(String[] a);
 public java.math.BigDecimal[] echoIndexTbl2(java.math.BigDecimal[] a);
 public IndexbytableintfVarrayTbl3 echoVarrayTbl4(IndexbytableintfVarrayTbl3 a);
 public IndexbytableintfNestedTbl4 echoVarrayTbl4(IndexbytableintfNestedTbl4 a);
}
```

In the generated code, the simple index-by table types, `index_tbl1` and `index_tb2`, are mapped to `String[]` and `BigDecimal[]` respectively. The nested table and varray table, however, are still mapped to custom JDBC types, because they are not index-by tables and their mappings are not affected by the `-plsqlindextable` setting.

The limitation of mapping PL/SQL index-by table to an array is that the table must be indexed by integer. If a PL/SQL package contains both tables indexed by integer and by `VARCHAR`, you cannot use the setting `-plsqlindexbytable=array` or `-plsqlindexbytable=`*`int`*. Otherwise the mapping for the table indexed by `VARCHAR` will encounter run-time errors. Instead, one must use `-plsqlindexbytable=custom`.

Mapping of the index-by table elements follows the JDBC type mappings. For example, with JDBC mapping, `SMALLINT` is mapped to the Java `int` type. Therefore, an index-by table of `SMALLINT` is mapped to `int[]`. The `-plsqlindexbytable=array` or `-plsqlindexbytable=`*`int`* setting will be ignored if Oracle mappings are turned on for numbers, that is, `-numbertypes=oracle`. The reason is that the Java array mapped to the index-by table must have string or numerical Java types as elements, while Oracle mappings map SQL numbers into `oracle.sql` types.

The Oracle JDBC drivers directly support PL/SQL scalar index-by tables with numeric or character elements. An index-by table with numeric elements can be mapped to the following Java array types:

- `int[]`
- `double[]`
- `float[]`
- `java.math.BigDecimal[]`
- `oracle.sql.NUMBER[]`

An index-by table with character elements can be mapped to the following Java array types:

- `String[]`
- `oracle.sql.CHAR[]`

In the following circumstances, you must convey certain information for an index-by table type, as described:

- Whenever you use the index-by table type in an `OUT` or `IN OUT` parameter, you must specify the maximum number of elements, which is otherwise optional. You can specify the maximum number of elements using the customary syntax for Java array allocation. For example, you could specify `int[100]` to denote a type that

can accommodate up to 100 elements or `oracle.sql.CHAR[20]` for up to 20 elements.

- For index-by tables with character elements, you can optionally specify the maximum size of an individual element, in bytes. This setting is defined using the SQL-like size syntax. For example, for an index-by table used for `IN` arguments, you could specify `String[](30)`. You could also specify `oracle.sql.CHAR[20](255)` for an index-by table of maximum length 20, the elements of which will not exceed 255 bytes each.

Use the JPublisher `-addtypemap` option to add instructions to the user type map to specify correspondences between PL/SQL types, which are scalar index-by tables, and the corresponding Java array types. The size hints that are given using the syntax outlined earlier are embedded into the generated SQLJ class and thus conveyed to JDBC at run time.

As an example, consider the following code fragment from the definition of the `INDEXBY` PL/SQL package in the `SCOTT` schema. Assume this is available in a file called `indexby.sql`.

```
CREATE OR REPLACE PACKAGE indexby AS

--  jpub.addtypemap=SCOTT.INDEXBY.VARCHAR_ARY:String[1000](4000)
--  jpub.addtypemap=SCOTT.INDEXBY.INTEGER_ARY:int[1000]
--  jpub.addtypemap=SCOTT.INDEXBY.FLOAT_ARY:double[1000]

 TYPE varchar_ary IS TABLE OF VARCHAR2(4000) INDEX BY BINARY_INTEGER;
 TYPE integer_ary IS TABLE OF INTEGER        INDEX BY BINARY_INTEGER;
 TYPE float_ary   IS TABLE OF NUMBER         INDEX BY BINARY_INTEGER;

 FUNCTION get_float_ary RETURN float_ary;
 PROCEDURE pow_integer_ary(x integer_ary, y OUT integer_ary);
 PROCEDURE xform_varchar_ary(x IN OUT varchar_ary);

END indexby;
/
CREATE OR REPLACE PACKAGE BODY indexby IS ...
/
```

The following are the required `-addtypemap` directives for mapping the three index-by table types:

```
-addtypemap=SCOTT.INDEXBY.VARCHAR_ARY:String[1000](4000)
-addtypemap=SCOTT.INDEXBY.INTEGER_ARY:int[1000]
-addtypemap=SCOTT.INDEXBY.FLOAT_ARY:double[1000]
```

Note that depending on the operating system shell you are using, you may have to quote options that contain square brackets `[...]` or parentheses `(...)`. You can avoid this by placing such options into a JPublisher properties file, as follows:

```
jpub.addtypemap=SCOTT.INDEXBY.VARCHAR_ARY:String[1000](4000)
jpub.addtypemap=SCOTT.INDEXBY.INTEGER_ARY:int[1000]
jpub.addtypemap=SCOTT.INDEXBY.FLOAT_ARY:double[1000]
```

> **See Also:** "Properties File Structure and Syntax" on page 6-51 and "Additional Entry to the User Type Map" on page 6-26

Additionally, as a feature of convenience, JPublisher directives in a properties file are recognized when placed behind a "--" prefix (two dashes), whereas any entry that

does not start with `"jpub."` or with `"-- jpub."` is ignored. So, you can place JPublisher directives into SQL scripts and reuse the same SQL scripts as JPublisher properties files. Thus, after invoking the `indexby.sql` script to define the `INDEXBY` package, you can now run JPublisher to publish this package as a Java class, `IndexBy`, as follows:

```
% jpub -u scott/tiger -s INDEXBY:IndexBy -props=indexby.sql
```

As mentioned previously, you can use this mapping of scalar index-by tables only with the Oracle JDBC drivers. If you are using another driver or if you want to create driver-independent code, then you must define SQL types that correspond to the index-by table types, as well as defining conversion functions that map between the two.

> **See Also:** "Type Mapping Support for PL/SQL RECORD and Index-By Table Types" on page 3-19

Type Mapping Support Through PL/SQL Conversion Functions

This section discusses the mechanism that JPublisher uses for supporting PL/SQL types in Java code, through PL/SQL conversion functions that convert between each PL/SQL type and a corresponding SQL type to allow access by JDBC.

In general, Java programs do not support the binding of PL/SQL-specific types. The only way you can use such types from Java is to use PL/SQL code to map them to SQL types, and then access these SQL types from Java. However, one exception is the scalar index-by table type.

JPublisher makes this task more convenient through the use of its type maps. For a particular PL/SQL type, specify the following information in a JPublisher type map entry:

- Name of the PL/SQL type, typically of the form:

  SCHEMA.PACKAGE.TYPE

- Name of the corresponding Java wrapper class
- Name of the SQL type that corresponds to the PL/SQL type

  You must be able to directly map this type to the Java wrapper type. For example, if the SQL type is `NUMBER`, then the corresponding Java type could be `int`, `double`, `Integer`, `Double`, `java.math.BigDecimal`, or `oracle.sql.NUMBER`. If the SQL type is an object type, then the corresponding Java type would be an object wrapper class that implements the `oracle.sql.ORAData` or `java.sql.SQLData` interface. The object wrapper class is typically generated by JPublisher.
- Name of a PL/SQL conversion function that maps the SQL type to the PL/SQL type
- Name of a PL/SQL conversion function that maps the PL/SQL type to the SQL type

The `-addtypemap` specification for this has the following form:

```
-addtypemap=plsql_type:java_type:sql_type:sql_to_plsql_fun:plsql_to_sql_fun
```

> **See Also:** "Type Map Options" on page 6-25

As an example, consider a type map entry for supporting the PL/SQL BOOLEAN type. It consists of the following specifications:

- Name of the PL/SQL type: BOOLEAN
- Specification to map it to Java boolean
- Corresponding SQL type: INTEGER

  JDBC considers boolean values as special numeric values.
- Name of the PL/SQL function that maps from SQL to PL/SQL: INT2BOOL

  The code for the function is:

```
FUNCTION int2bool(i INTEGER) RETURN BOOLEAN IS
BEGIN IF i IS NULL THEN RETURN NULL;
      ELSE RETURN i<>0;
      END IF;
END int2bool;
```

- Name of the PL/SQL function that maps from PL/SQL to SQL: BOOL2INT

  The code for the function is:

```
FUNCTION bool2int(b BOOLEAN) RETURN INTEGER IS
BEGIN IF b IS NULL THEN RETURN NULL;
      ELSIF b THEN RETURN 1;
      ELSE RETURN 0;
      END IF;
END bool2int;
```

You can put all this together in the following type map entry:

```
-addtypemap=BOOLEAN:boolean:INTEGER:INT2BOOL:BOOL2INT
```

Such a type map entry assumes that the SQL type, the Java type, and both conversion functions have been defined in SQL, Java, and PL/SQL, respectively. Note that there is already an entry for PL/SQL BOOLEAN in the JPublisher default type map. If you want to try the preceding type map entry, you will have to override the default type map. You can use the JPublisher -defaulttypemap option to accomplish this, as follows:

```
% jpub -u scott/tiger -s SYS.SQLJUTL:SQLJUtl
       -defaulttypemap=BOOLEAN:boolean:INTEGER:INT2BOOL:BOOL2INT
```

Notes:

- In some cases, such as with INT2BOOL and BOOL2INT in the preceding example, JPublisher has conversion functions that are predefined, typically in the SYS.SQLJUTL package. In other cases, such as for RECORD types and index-by table types, JPublisher generates conversion functions during execution.
- Although this manual describes conversions as mapping between SQL and PL/SQL types, there is no intrinsic restriction to PL/SQL in this approach. You could also map between different SQL types. In fact, this is done in the JPublisher default type map to support SQL INTERVAL types, which are mapped to VARCHAR2 values and back.

Be aware that under some circumstances, PL/SQL wrapper functions are also created by JPublisher. Each wrapper function wraps a stored procedure that uses PL/SQL types. It calls this original stored procedure and processes its PL/SQL input or output through the appropriate conversion functions so that only the corresponding SQL types are exposed to Java. The following JPublisher options control how JPublisher creates code for invocation of PL/SQL stored procedures that use PL/SQL types, including the use of conversion functions and possibly the use of wrapper functions:

- `-plsqlpackage=`*`plsql_package`*

  This option determines the name of the PL/SQL package into which JPublisher generates the PL/SQL conversion functions: a function to convert each unsupported PL/SQL type to the corresponding SQL type and a function to convert from each corresponding SQL type back to the PL/SQL type. Optionally, depending on how you set the `-plsqlmap` option, the package also contains wrapper functions for the original stored procedures, with each wrapper function invoking the appropriate conversion function.

  If you do not specify a package name, then JPublisher uses `JPUB_PLSQL_WRAPPER`.

- `-plsqlfile=`*`plsql_wrapper_script`*`,`*`plsql_dropper_script`*

  This option determines the name of the wrapper script and dropper script that JPublisher creates. The wrapper script creates necessary SQL types that map to unsupported PL/SQL types and also creates the PL/SQL package. The dropper script drops these SQL types and the PL/SQL package.

  If the files already exist, then they will be overwritten. If you do not specify any file names, then JPublisher will write to the files named `plsql_wrapper.sql` and `plsql_dropper.sql`.

- `-plsqlmap=`*`flag`*

  This option specifies whether JPublisher generates wrapper functions for stored procedures that use PL/SQL types. Each wrapper function calls the corresponding stored procedure and the appropriate PL/SQL conversion functions for PL/SQL input or output of the stored procedure. Only the corresponding SQL types are exposed to Java. The *`flag`* setting can be any of the following:

  - `true`

    This is the default setting. JPublisher generates PL/SQL wrapper functions only as needed. For any given stored procedure, if the Java code to call it and convert its PL/SQL types directly is simple enough, and if PL/SQL types are used only as `IN` parameters or for the function return, then the generated code calls the stored procedure directly instead. The code then processes the PL/SQL input or output through the appropriate conversion functions.

    If a PL/SQL type is used as an `OUT` or `IN OUT` parameter, then wrapper functions are required, because conversions between PL/SQL and SQL representations may be necessary either before or after calling the original stored procedure.

  - `false`

    JPublisher does not generate PL/SQL wrapper functions. If it encounters a PL/SQL type in a signature that cannot be supported by a direct call and conversion, then it skips the generation of Java code for the particular stored procedure.

  - `always`

JPublisher generates a PL/SQL wrapper function for every stored procedure that uses a PL/SQL type. This setting is useful for generating a *proxy* PL/SQL package that complements an original PL/SQL package, providing JDBC-accessible signatures for those functions or procedures that were not accessible through JDBC in the original package.

See Also: "Direct Use of PL/SQL Conversion Functions Versus Use of Wrapper Functions" on page 3-22 and "PL/SQL Code Generation Options" on page 6-37

Type Mapping Support for PL/SQL RECORD and Index-By Table Types

JPublisher automatically publishes a PL/SQL `RECORD` type whenever it publishes a PL/SQL stored procedure or function that uses that type as an argument or return type. The same is true for PL/SQL index-by table types. This is the only way that a `RECORD` type or index-by table type can be published. There is no way to explicitly request any such types to be published through JPublisher option settings.

Note: The following are limitations to the JPublisher support for PL/SQL `RECORD` and index-by table types:

- An intermediate wrapper layer is required to map a `RECORD` or index-by-table argument to a SQL type that JDBC can support. In addition, JPublisher cannot fully support the semantics of index-by tables. An index-by table is similar in structure to a Java hashtable, but information is lost when JPublisher maps this to a SQL `TABLE` type.
- If you use the JDBC OCI driver and require only the publishing of scalar index-by tables, then you can use the direct mapping between Java and these types.

The following sections demonstrate JPublisher support for PL/SQL `RECORD` types and index-by table types:

- Sample Package for RECORD Type and Index-By Table Type Support
- Support for RECORD Types
- Support for Index-By Table Types

Sample Package for RECORD Type and Index-By Table Type Support

The following PL/SQL package is used to illustrate JPublisher support for PL/SQL `RECORD` and index-by table types:

```
CREATE OR REPLACE PACKAGE company IS
  TYPE emp_rec IS RECORD (empno NUMBER, ename VARCHAR2(10));
  TYPE emp_tbl IS TABLE OF emp_rec INDEX BY binary_integer;
  PROCEDURE set_emp_rec(er emp_rec);
  FUNCTION get_emp_rec(empno number) RETURN emp_rec;
  FUNCTION get_emp_tbl RETURN emp_tbl;
END;
```

This package defines a PL/SQL `RECORD` type, `EMP_REC`, and a PL/SQL index-by table type, `EMP_TBL`. Use the following command to publish the `COMPANY` package:

```
% jpub -u scott/tiger -s COMPANY:Company -plsqlpackage=WRAPPER1
  -plsqlfile=wrapper1.sql,dropper1.sql
```

The JPublisher output is:

```
SCOTT.COMPANY
SCOTT."COMPANY.EMP_REC"
SCOTT."COMPANY.EMP_TBL"
J2T-138, NOTE: Wrote PL/SQL package WRAPPER1 to file wrapper1.sql.
Wrote the dropping script to file dropper1.sql
```

In this example, JPublisher generates `Company.java` for the Java wrapper class for the `COMPANY` package, as well as the following SQL and Java entities:

- The `wrapper1.sql` script that creates the SQL types corresponding to the PL/SQL `RECORD` and index-by table types, and also creates the conversion functions between the SQL types and the PL/SQL types
- The `dropper1.sql` script that removes the SQL types and conversion functions created by `wrapper1.sql`
- The `CompanyEmpRec.java` source file for the Java wrapper class for the SQL object type that is generated for the PL/SQL `RECORD` type
- The `CompanyEmpTbl.java` source file for the Java wrapper class for the SQL collection type that is generated for the PL/SQL index-by table type

Support for RECORD Types

This section continues the example from Sample Package for RECORD Type and Index-By Table Type Support. For the PL/SQL `RECORD` type, `EMP_REC`, JPublisher generates the corresponding `COMPANY_EMP_REC` SQL object type. JPublisher also generates the conversion functions between the two. In this example, the following is generated in `wrapper1.sql` for `EMP_REC`:

```
CREATE OR REPLACE TYPE COMPANY_EMP_REC AS OBJECT (
                      EMPNO NUMBER(22),
                      ENAME VARCHAR2(10)
);
/
-- Declare package containing conversion functions between SQL and PL/SQL types
CREATE OR REPLACE PACKAGE WRAPPER1 AS
   -- Declare the conversion functions the PL/SQL type COMPANY.EMP_REC
        FUNCTION PL2COMPANY_EMP_REC(aPlsqlItem COMPANY.EMP_REC)
        RETURN COMPANY_EMP_REC;
        FUNCTION COMPANY_EMP_REC2PL(aSqlItem COMPANY_EMP_REC)
        RETURN COMPANY.EMP_REC;
END WRAPPER1;
/
```

In addition, JPublisher publishes the `COMPANY_EMP_REC` SQL object type into the `CompanyEmpRec.java` Java source file.

Once the PL/SQL `RECORD` type is published, you can add the mapping to the type map. The following is an entry in a sample JPublisher properties file, `done.properties`:

```
jpub.addtypemap=SCOTT.COMPANY.EMP_REC:CompanyEmpRec:COMPANY_EMP_REC:
WRAPPER1.COMPANY_EMP_REC2PL:WRAPPER1.PL2COMPANY_EMP_REC
```

Use this type map entry whenever you publish a package or type that refers to the `RECORD` type, `EMP_REC`. For example, the following JPublisher invocation uses `done.properties` with this type map entry:

```
% jpub -u scott/tiger -p done.properties -s COMPANY -plsqlpackage=WRAPPER2
       -plsqlfile=wrapper2.sql,dropper2.sql
```

The JPublisher output is:

```
SCOTT.COMPANY
SCOTT."COMPANY.EMP_TBL"
J2T-138, NOTE: Wrote PL/SQL package WRAPPER2 to file wrapper2.sql.
Wrote the dropping script to file dropper2.sql
```

Support for Index-By Table Types

This section continues the example from Sample Package for RECORD Type and Index-By Table Type Support.

To support an index-by table type, a SQL collection type must be defined that permits conversion to and from the PL/SQL index-by table type. JPublisher also supports PL/SQL nested tables and VARRAYs in the same fashion. Therefore, JPublisher generates the same code for the following three definitions of `EMP_TBL`:

```
TYPE emp_tbl IS TABLE OF emp_rec INDEX BY binary_integer;
TYPE emp_tbl IS TABLE OF emp_rec;
TYPE emp_tbl IS VARRAY OF emp_rec;
```

For the PL/SQL index-by table type `EMP_TBL`, JPublisher generates a SQL collection type, and conversion functions between the index-by table type and the SQL collection type.

In addition to what was shown for the `RECORD` type earlier, JPublisher generates the following:

```
-- Declare the SQL type for the PL/SQL type COMPANY.EMP_TBL
CREATE OR REPLACE TYPE COMPANY_EMP_TBL AS TABLE OF COMPANY_EMP_REC;
/
-- Declare package containing conversion functions between SQL and PL/SQL types
CREATE OR REPLACE PACKAGE WRAPPER1 AS
   -- Declare the conversion functions for the PL/SQL type COMPANY.EMP_TBL
        FUNCTION PL2COMPANY_EMP_TBL(aPlsqlItem COMPANY.EMP_TBL)
        RETURN COMPANY_EMP_TBL;
        FUNCTION COMPANY_EMP_TBL2PL(aSqlItem COMPANY_EMP_TBL)
        RETURN COMPANY.EMP_TBL;
...
END WRAPPER1;
```

JPublisher further publishes the SQL collection type into `CompanyEmpTbl.java`.

As with a PL/SQL `RECORD` type, once a PL/SQL index-by table type is published, the published result, including the Java wrapper classes, the SQL collection type, and the conversion functions, can be used in the future for publishing PL/SQL packages involving that PL/SQL index-by table type. For example, if you add the following entry into a properties file that you use in invoking JPublisher, say `done.properties`, then JPublisher will use the provided type map and avoid republishing that index-by table type:

```
jpub.addtypemap=SCOTT.COMPANY.EMP_TBL:CompanyEmpTbl:COMPANY_EMP_TBL:
WRAPPER1.COMPANY_EMP_TBL2PL:WRAPPER1.PL2COMPANY_EMP_TBL
```

> **See Also:** "JPublisher User Type Map and Default Type Map" on page 3-5

Direct Use of PL/SQL Conversion Functions Versus Use of Wrapper Functions

In generating Java code to invoke a stored procedure that uses a PL/SQL type, JPublisher can use either of the following modes of operation:

- Invoke the stored procedure directly, which processes the PL/SQL input or output through the appropriate conversion functions.
- Invoke a PL/SQL wrapper function, which in turn calls the stored procedure and processes its PL/SQL input or output through the appropriate conversion functions. The wrapper function that is generated by JPublisher uses the corresponding SQL types for input or output.

The `-plsqlmap` option determines whether JPublisher uses the first mode, the second mode, or possibly either mode, depending on circumstances.

See Also: "Generation of PL/SQL Wrapper Functions" on page 6-38

As an example, consider the `SCOTT.COMPANY.GET_EMP_TBL` PL/SQL stored procedure that returns the `EMP_TBL` PL/SQL index-by table type. Assume that the `COMPANY` package, introduced in "Sample Package for RECORD Type and Index-By Table Type Support" on page 3-19, is processed by JPublisher through the following command:

```
% jpub -u scott/tiger -s COMPANY:Company -plsqlpackage=WRAPPER1
       -plsqlfile=wrapper1.sql,dropper1.sql -plsqlmap=false
```

The JPublisher output is:

```
SCOTT.COMPANY
SCOTT."COMPANY.EMP_REC"
SCOTT."COMPANY.EMP_TBL"
J2T-138, NOTE: Wrote PL/SQL package WRAPPER1 to file wrapper1.sql.
Wrote the dropping script to file dropper1.sql
```

With this command, JPublisher creates the following:

- SQL object type `COMPANY_EMP_REC` to map to the PL/SQL `RECORD` type `EMP_REC`
- SQL collection type `COMPANY_EMP_TBL` to map to the PL/SQL index-by table type `EMP_TBL`
- Java classes to map to `COMPANY`, `COMPANY_EMP_REC`, and `COMPANY_EMP_TBL`
- PL/SQL package `WRAPPER1`, which includes the PL/SQL conversion functions to convert between the PL/SQL index-by table type and the SQL collection type

In this example, assume that the conversion function `PL2COMPANY_EMP_TBL` converts from the PL/SQL `EMP_TBL` type to the SQL `COMPANY_EMP_TBL` type. Because of the setting `-plsqlmap=false`, no wrapper functions are created. The stored procedure is called with the following JDBC statement in generated Java code:

```
conn.prepareOracleCall =
("BEGIN :1 := WRAPPER1.PL2COMPANY_EMP_TBL(SCOTT.COMPANY.GET_EMP_TBL()) \n; END;");
```

`SCOTT.COMPANY.GET_EMP_TBL` is called directly, with its `EMP_TBL` output being processed through the `PL2COMPANY_EMP_TBL` conversion function to return the desired `COMPANY_EMP_TBL` SQL type.

By contrast, if you run JPublisher with the setting `-plsqlmap=always`, then `WRAPPER1` also includes a PL/SQL wrapper function for every PL/SQL stored procedure that uses a PL/SQL type. In this case, for any given stored procedure, the

generated Java code calls the wrapper function instead of the stored procedure. The wrapper function, in this example `WRAPPER1.GET_EMP_TBL`, calling the original stored procedure and processing its output through the conversion function is as follows:

```
FUNCTION  GET_EMP_TBL()
   BEGIN
      RETURN WRAPPER1.PL2COMPANY_EMP_TBL(SCOTT.COMPANY.GET_EMP_TBL())
   END;
```

In the generated Java code, the JDBC statement calling the wrapper function is:

```
conn.prepareOracleCall("BEGIN :1=SCOTT.WRAPPER1.GET_EMP_TBL() \n; END;");
```

If `-plsqlmap=true`, then JPublisher uses direct calls to the original stored procedure wherever possible. However, in the case of any stored procedure for which the Java code for direct invocation and conversion is too complex or any stored procedure that uses PL/SQL types as `OUT` or `IN OUT` parameters, JPublisher generates a wrapper function and calls that function in the generated code.

Other Alternatives for Data Types Unsupported by JDBC

The preceding sections describe the mechanisms that JPublisher employs to access PL/SQL types unsupported by JDBC. As an alternative to using JPublisher in this way, you can try one of the following:

- Rewrite the PL/SQL method to avoid using the type
- Write an anonymous block that does the following:
  - Converts input types that JDBC supports into the input types used by the PL/SQL stored procedure
  - Converts output types used by the PL/SQL stored procedure into output types that JDBC supports

JPublisher Styles and Style Files

JPublisher style files allow you to specify Java-to-Java type mappings. This is to ensure that generated classes can be used in Web services. As an example, `CLOB` types, such as `java.sql.Clob` and `oracle.sql.CLOB`, cannot be used in Web services, but the data can be used if it is converted to a type that is supported by Web services, such as `java.lang.String`. JPublisher must generate user subclasses to implement its use of style files and Java-to-Java type transformations.

> **See Also:** "JPublisher-Generated Subclasses for Java-to-Java Type Transformations" on page 4-16

Typically, style files are provided by Oracle, but there may be situations in which you may want to edit or create your own.

The following sections discuss features and usage of styles and style files:

- Style File Specifications and Locations
- Style File Format
- Summary of Key Java-to-Java Type Mappings in Oracle Style Files
- Use of Multiple Style Files

Style File Specifications and Locations

Use the JPublisher `-style` option to specify the base name of a style file:

```
-style=stylename
```

Based on the *`stylename`* you specify, JPublisher looks for a style file as follows, and uses the first file that it finds:

1. It looks for the following resource in the classpath:

   ```
   /oracle/jpub/mesg/stylename.properties
   ```

2. It takes *`stylename`* as a resource name, possibly qualified, and looks for the following in the classpath:

   ```
   /stylename-dir/stylename-base.properties
   ```

3. It takes *`stylename`* as a name, possibly qualified, and looks for the following file in the current directory:

   ```
   stylename.properties
   ```

   In this case, *`stylename`* can optionally include a directory path. If you use the setting `-style=mydir/foo`, for example, then JPublisher looks for `mydir/foo.properties` relative to the current directory.

If no matching file is found, JPublisher generates an exception.

See Also: "Style File for Java-to-Java Type Mappings" on page 6-25

As an example of the first scenario, if the resource `/oracle/jpub/mesg/webservices.properties` exists in *`ORACLE_HOME`*`/sqlj/lib/translator.jar` and `translator.jar` is found in the classpath, then the `-style=webservices` setting uses `/oracle/jpub/mesg/webservices.properties` from `translator.jar`, even if there is a `webservices.properties` file in the current directory.

However, if you specify `-style=mystyle` and a `mystyle.properties` resource is *not* found in `/oracle/jpub/mesg`, but there is a `mystyle.properties` file in the current directory, then that is used.

Note: Oracle currently provides three style files:

```
/oracle/jpub/mesg/webservices-common.properties
/oracle/jpub/mesg/webservices10.properties
/oracle/jpub/mesg/webservices9.properties
```

These are in the `translator.jar` file, which must be included in your classpath. Each file maps Oracle JDBC types to Java types supported by Web services. Note that the `webservices-common.properties` file is for general use and is included by both `webservices10.properties` and `webservices9.properties`.

To use Web services in Oracle Database 10*g*, specify the following style file:

```
-style=webservices10
```

To use Web services in Oracle9*i*, specify `-style=webservices9`.

Style File Format

The key portion of a style file is the `TRANSFORMATION` section. This section comprises everything between the `TRANSFORMATION` tag and `END_TRANSFORMATION` tag. It describes the type transformations, or Java-to-Java mappings, to be applied to types used for object attributes or in method signatures.

For convenience, there is an `OPTIONS` section in which you can specify any other JPublisher option settings. Because of this section, a style file can replace the functionality of any other JPublisher properties file, in addition to specifying mappings.

This section covers the following topics:

- Style File TRANSFORMATION Section
- Style File OPTIONS Section

Note: The following details about style files are provided for general information only and are subject to change.

Style File TRANSFORMATION Section

This section provides a template for a style file `TRANSFORMATION` section, with comments. Within the `TRANSFORMATION` section, there is a `MAPPING` section for each mapping that you specify. The `MAPPING` section starts at a `MAPPING` tag and ends with an `END_MAPPING` tag. Each `MAPPING` section includes a number of subtags with additional information. In the `MAPPING` section, the `SOURCETYPE` and `TARGETTYPE` tags are the required subtags. Within each `TARGETTYPE` section, you should generally provide information for at least the `RETURN`, `IN`, and `OUT` cases, using the corresponding tags. The following code illustrates the structure of a typical `TRANSFORMATION` section:

```
TRANSFORMATION

 IMPORT
 # Packages to be imported by the generated classes
 END_IMPORT
```

```
# THE FOLLOWING OPTION ONLY APPLIES TO PL/SQL PACKAGES
# This interface should be implemented/extended by
# the methods in the user subclasses and interfaces
# This option takes no effect when subclass is not generated.
SUBCLASS_INTERFACE java_interface

# THE FOLLOWING OPTION ONLY APPLIES TO PL/SQL PACKAGES
# Each method in the interface and the user subclass should
# throw this exception (the default SQLException will be caught
# and re-thrown as an exception specified here)
# This option takes no effect when subclass is not generated.
SUBCLASS_EXCEPTION Java_exception_type

STATIC
# Any code provided here is inserted at the
# top level of the generated subclass regardless
# of the actual types used.
END_STATIC

# Enumerate as many MAPPING sections as needed.

MAPPING
SOURCETYPE Java_source_type
# Can be mapped to several target types.
TARGETTYPE Java_target_type

# With CONDITION specified, the source-to-target
# mapping is carried out only when the listed Java
# classes are present during publishing.
# The CONDITION section is optional.
CONDITION list_of_java_classes

IN
# Java code for performing the transformation
# from source type argument %1 to the target
# type, assigning it to %2.
END_IN
IN_AFTER_CALL
# Java code for processing IN parameters
# after procedure call.
END_IN_AFTER_CALL
OUT
# Java code for performaing the transformation
# from a target type instance %2 to the source
# type, assigning it to %1.
END_OUT
RETURN
# Java code for performing the transformation
# from source type argument %1 to the target
# type and returning the target type.
END_RETURN

# Include the code given by a DEFINE...END_DEFINE block
# at the end of this template file.
USE defined_name

# Holder for OUT/INOUT of the type defined by SOURCETYPE.
HOLDER Java_holder_type
END_TARGETTYPE
```

```
 # More TARGETTYPE sections, as needed

 END_MAPPING

 DEFAULT_HOLDER
 # JPublisher will generate holders for types that do
 # not have HOLDER entries defined in this template.
 # This section includes a template for class definitions
 # from which JPublisher will generate .java files for
 # holder classes.
 END_DEFAULT_HOLDER

 # More MAPPING sections, as needed

 DEFINE defined_name
 # Any code provided here is inserted at the
 # top level of the generated class if the
 # source type is used.
 END_DEFINE
 # More DEFINE sections, as needed

END_TRANSFORMATION
```

Notes:

- Style files use `ISO8859_1` encoding. Any characters that cannot be represented directly in this encoding must be represented in Unicode escape sequences.
- It is permissible to have multiple `MAPPING` sections with the same `SOURCETYPE` specification. For argument type, JPublisher uses the last of these `MAPPING` sections that it encounters.

See Also: "Passing Output Parameters in JAX-RPC Holders" on page 4-4 for a discussion of holders

Style File OPTIONS Section

For convenience, you can specify any desired JPublisher option settings in the `OPTIONS` section of a style file, in the standard format for JPublisher properties files. The syntax for the same is as follows:

```
OPTIONS
 # Comments
 jpub.option1=value1
 jpub.option2=value2
 ...
END_OPTIONS
```

Summary of Key Java-to-Java Type Mappings in Oracle Style Files

The Oracle style files `webservices-common.properties`, `webservices9.properties`, and `webservices10.properties`, through their `SOURCETYPE` and `TARGETTYPE` specifications, have a number of important Java-to-Java type mappings to support Web services and `REF CURSOR` mappings. These mappings are summarized in Table 3–2.

Table 3–2 Summary of Java-to-Java Type Mappings in Oracle Style Files

Source Type	Target Type
`oracle.sql.NString`	`java.lang.String`
`oracle.sql.CLOB`	`java.lang.String`
`oracle.sql.BLOB`	`byte[]`
`oracle.sql.BFILE`	`byte[]`
`java.sql.Timestamp`	`java.util.Date`
`java.sql.ResultSet`	`oracle.jdbc.rowset.OracleWebRowSet`
	`org.w3c.dom.Document`
	`javax.xml.transform.Source`
`oracle.sql.SimpleXMLType`	`java.lang.String` (`webservices-common`)
	`org.w3c.dom.DocumentFragment` (`webservices9`)
	`javax.xml.transform.Source` (`webservices10`)

The `webservices9` and `webservices10` files include `webservices-common` before specifying mappings specific to these files. For `SimpleXMLType`, note that `DocumentFragment` overrides `String` if you set `-style=webservices9` and `Source` overrides `String`, if you set `-style=webservices10`.

> **See Also:** "REF CURSOR Types and Result Sets Mapping" on page 3-7

Use of Multiple Style Files

JPublisher allows multiple `-style` options on the command line, with the following behavior:

- The `OPTIONS` sections are concatenated.
- The `TRANSFORMATION` sections are concatenated, except that the entries in the `MAPPING` sections are overridden, as applicable. A `MAPPING` entry from a style file specified later on the command line overrides a `MAPPING` entry with the same `SOURCETYPE` specification from a style file specified earlier on the command line.

This functionality is useful if you want to overwrite type mappings defined earlier or add new type mappings. For example, if you want to map `SYS.XMLTYPE` to `java.lang.String`, then you can append the setting `-style=xml2string` to the JPublisher command line. This example assumes that the `./xml2string.properties` style file will be accessed. This style file is defined as follows:

```
OPTIONS
 jpub.defaulttypemap=SYS.XMLTYPE:oracle.sql.SimpleXMLType
END_OPTIONS
TRANSFORM
MAPPING
SOURCETYPE oracle.sql.SimpleXMLType
TARGETTYPE java.lang.String
# SimpleXMLType => String
OUT
%2 = null;
if (%1!=null) %2=%1.getstringval();
END_OUT
# String => SimpleXMLType
```

```
IN
%1 = null;
if (%2!=null)
{
  %1 = new %p.%c(_getConnection());
  %1 = %1.createxml(%2);
}
END_IN
END_TARGETTYPE
END_MAPPING
END_TRANSFORM
```

Continuing this example, assume the following PL/SQL stored procedure definition:

```
PROCEDURE foo (arg XMLTYPE);
```

JPublisher maps this as follows in the base class:

```
void foo (arg oracle.sql.SimpleXMLType);
```

And JPublisher maps it as follows in the user subclass:

```
void foo (arg String);
```

Note: By default, JPublisher maps `SYS.XMLTYPE` to `oracle.xdb.XMLType`.

4

Generated Classes and Interfaces

This chapter describes the classes, interfaces, and subclasses that JPublisher generates in the following sections:

Treatment of Output Parameters

Stored procedures called through Java Database Connectivity (JDBC) do not pass parameters in the same way as ordinary Java methods. This affects the code that you write when you call a wrapper method that JPublisher generates.

When you call an ordinary Java method, parameters that are Java objects are passed as object references. The method can modify the object.

However, when you call a stored procedure through JDBC, a copy of each parameter is passed to the stored procedure. If the procedure modifies any parameters, then a copy of the modified parameter is returned to the caller. Therefore, the *before* and *after* values of a modified parameter appear in separate objects.

A wrapper method that JPublisher generates contains JDBC statements to call the corresponding stored procedure. The parameters to the stored procedure, as declared in your `CREATE TYPE` or `CREATE PACKAGE` declaration, have the following possible parameter modes: `IN`, `OUT`, and `IN OUT`. Parameters that are `IN OUT` or `OUT` are returned to the wrapper method in newly created objects. These new values must be returned to the caller somehow, but assignment to the formal parameter within the wrapper method does not affect the actual parameter visible to the caller.

In Java, there are no `OUT` or `IN OUT` designations, but values can be returned through holders. In JPublisher, you can specify one of the following alternatives for holders that handle PL/SQL `OUT` or `IN OUT` parameters:

- Arrays
- Java API for XML-based Remote Procedure Call (JAX-RPC) holder types
- Function returns

The `-outarguments` option enables you to specify which mechanism to use. This feature is particularly useful for Web services.

See Also: "Holder Types for Output Arguments" on page 6-34

The following sections describe the three mechanisms:

- Passing Output Parameters in Arrays
- Passing Output Parameters in JAX-RPC Holders
- Passing Output Parameters in Function Returns

Passing Output Parameters in Arrays

One way to solve the problem of returning output values in Java is to pass an `OUT` or `IN OUT` parameter to the wrapper method in a single-element array. Think of the array as a container that holds the parameter. This mechanism works as follows:

1. You assign the *before* value of the parameter to element `[0]` of an array.
2. You pass the array to your wrapper method.
3. The wrapper method assigns the *after* value of the parameter to element `[0]` of the array.
4. After running the method, you extract the *after* value from the array.

A setting of `-outarguments=array`, which is the default, instructs JPublisher to use this single-element array mechanism to publish any `OUT` or `IN OUT` argument.

For example:

```
Person [] pa = {p};
x.f(pa);
p = pa[0];
```

Assume that `x` is an instance of a JPublisher-generated class that has the `f()` method, which is a wrapper method for a stored procedure that uses a SQL `PERSON` object as an `IN OUT` parameter. The `PERSON` type maps to the `Person` Java class. `p` is a `Person` instance, and `pa[]` is a single-element `Person` array.

This mechanism for passing `OUT` or `IN OUT` parameters requires you to add a few extra lines of code to your program for each parameter. As another example, consider the PL/SQL function created by the following SQL*Plus command:

```
SQL> CREATE OR REPLACE FUNCTION g (
            a0 NUMBER,
            a1 OUT NUMBER,
            a2 IN OUT NUMBER,
            a3 CLOB,
            a4 OUT CLOB,
            a5 IN OUT CLOB)
     RETURN CLOB IS
     BEGIN
            RETURN NULL;
     END;
```

With `-outarguments=array`, this is published as follows:

```
public oracle.sql.CLOB g (
     java.math.BigDecimal a0,
     java.math.BigDecimal a1[],
```

```
java.math.BigDecimal a2[],
oracle.sql.CLOB a3,
oracle.sql.CLOB a4[],
oracle.sql.CLOB a5[])
```

Problems similar to those described earlier arise when the `this` object of an instance method is modified.

The `this` object is an additional parameter, which is passed in a different way. Its mode, as declared in the `CREATE TYPE` statement, may be `IN` or `IN OUT`. If you do not explicitly declare the mode of the `this` object, then its mode is `IN OUT`, if the stored procedure does not return a result, or `IN`, if it does.

If the mode of the `this` object is `IN OUT`, then the wrapper method must return the new value of `this`. The code generated by JPublisher implements this functionality in different ways, depending on the situation, as follows:

- For a stored procedure that does not return a result, the new value of `this` is returned as the result of the wrapper method.

  As an example, assume that the SQL object type `MYTYPE` has the following member procedure:

  ```
  MEMBER PROCEDURE f1(y IN OUT INTEGER);
  ```

  Also, assume that JPublisher generates a corresponding Java class, `MyJavaType`. This class defines the following method:

  ```
  MyJavaType f1(int[] y)
  ```

  The `f1()` method returns the modified `this` object value as a `MyJavaType` instance.

- For a stored function, which is a stored procedure that returns a result, the wrapper method returns the result of the stored function as its result. The new value of `this` is returned in a single-element array, passed as an extra argument, which is the last argument, to the wrapper method.

  Assume that the SQL object type `MYTYPE` has the following member function:

  ```
  MEMBER FUNCTION f2(x IN INTEGER) RETURNS VARCHAR2;
  ```

  Then the corresponding Java class, `MyJavaType`, defines the following method:

  ```
  String f2(int x, MyJavaType[] newValue)
  ```

  The `f2()` method returns the `VARCHAR2` value as a Java string and the modified `this` object value as an array element in the `MyJavaType` array.

> **Note:** For PL/SQL static procedures or functions, JPublisher generates instance methods, and not static methods, in the wrapper class. This is the logistic for associating a database connection with each wrapper class instance. The connection instance is used in initializing the wrapper class instance so that you are not subsequently required to explicitly provide a connection or connection context instance when calling wrapper methods.

Passing Output Parameters in JAX-RPC Holders

The JAX-RPC specification explicitly specifies holder classes in the `javax.xml.rpc.holders` package for the Java mapping of simple XML data types and other types. Typically, `Holder` is appended to the type name for the holder class name. For example, `BigDecimalHolder` is the holder class for `BigDecimal`.

Given a setting of `-outarguments=holder`, JPublisher uses holder instances to publish `OUT` and `IN OUT` arguments from stored procedures. Holder settings are specified in a JPublisher style file. The settings are specified in the `HOLDER` subtag inside the `TARGETTYPE` section for appropriate mapping. If no holder class is specified, then JPublisher chooses one according to defaults.

See Also: "JPublisher Styles and Style Files" on page 3-23

For general information about JAX-RPC and holders, refer to the *Java API for XML-based RPC, JAX-RPC 1.0* specification, available at:

`http://jcp.org/aboutJava/communityprocess/final/jsr101/index.html`

As an example, consider the PL/SQL function created by the following SQL*Plus command:

```
SQL> CREATE OR REPLACE FUNCTION g (
            a0 NUMBER,
            a1 OUT NUMBER,
            a2 IN OUT NUMBER,
            a3 CLOB,
            a4 OUT CLOB,
            a5 IN OUT CLOB)
     RETURN CLOB IS
     BEGIN
            RETURN NULL;
     END;
```

Assume that the `webservices10` style file contains an entry for `-outarguments=holder` and the following JPublisher command is used to publish the function, `g()`:

```
% jpub -u scott/tiger  -s toplevel"(g)":ToplevelG -style=webservices10
```

The published interface is:

```
public java.lang.String g
              (java.math.BigDecimal a0,
               javax.xml.rpc.holders.BigDecimalHolder _xa1_out_x,
               javax.xml.rpc.holders.BigDecimalHolder _xa2_inout_x,
               java.lang.String a3,
               javax.xml.rpc.holders.StringHolder _xa4_out_x,
               javax.xml.rpc.holders.StringHolder _xa5_inout_x)
throws java.rmi.RemoteException;
```

In this case, there is an extra level of abstraction. Because `oracle.sql.CLOB` is not supported by Web services, it is mapped to `String`, the JAX-RPC holder class for which is `StringHolder`.

See Also: "JPublisher-Generated Subclasses for Java-to-Java Type Transformations" on page 4-16

Passing Output Parameters in Function Returns

You can use the `-outarguments=return` setting as a workaround for supporting method signatures in Web services that do not use JAX-RPC holder types or arrays. If there is no support for JAX-RPC holders, the `-outarguments=return` setting allows `OUT` or `IN OUT` data to be returned in function results.

Consider the PL/SQL function created by the following SQL*Plus command:

```
SQL> CREATE OR REPLACE FUNCTION g (
            a0 NUMBER,
            a1 OUT NUMBER,
            a2 IN OUT NUMBER,
            a3 CLOB,
            a4 OUT CLOB,
            a5 IN OUT CLOB)
     RETURN CLOB IS
     BEGIN
            RETURN NULL;
     END;
```

Assume the following JPublisher command to publish the function, `g()`. Although the `webservices10` style file specifies `-outarguments=holder`, the `-outarguments=return` setting comes after the `-style` setting and, therefore, takes precedence.

```
% jpub -u scott/tiger  -s toplevel"(g)":ToplevelG -style=webservices10
       -outarguments=return
```

The JPublisher output is as follows:

```
SCOTT.top_level_scope
ToplevelGUser_g_Out
```

The JPublisher output acknowledges that it is processing the `SCOTT` top level and also indicates the creation of the `ToplevelGUser_g_Out` Java class to support output values of the `g()` function through return data.

Notes:

- The `_g_Out` appended to the user class name is according to the JPublisher naming convention used when creating a class to contain the output data in the scenario of passing output parameters in function returns. The `_g` reflects the name of the function being processed and the `_Out` reflects the `OUT` modifier in the corresponding PL/SQL call specification. Therefore, `ToplevelGUser_g_Out` is the Java type created for the output data of the `g()` method in the `ToplevelGUser` class. The user class name is according to the naming convention specified in the `webservices10` style file.
- Typically, JPublisher output reflects only the names of SQL or PL/SQL entities being processed, but there is no such entity that directly corresponds to `ToplevelGUser_g_Out`.

JPublisher generates the following interface to take input parameters and return output parameters:

```
public ToplevelGUser_g_Out g
           (java.math.BigDecimal a0,
```

```
            java.math.BigDecimal xxa2_inoutxx,
            java.lang.String a3,
            java.lang.String xxa5_inoutxx)
throws java.rmi.RemoteException;
```

JPublisher generates the `TopLevelGUser_g_Out` class as follows:

```
public class ToplevelGUser_g_Out
{
  public ToplevelGUser_g_Out() { }
  public java.math.BigDecimal getA1Out()  { return a1_out; }
  public void setA1Out(java.math.BigDecimal a1_out) { this.a1_out = a1_out; }
  public java.math.BigDecimal getA2Inout()   { return a2_inout; }
  public void setA2Inout(java.math.BigDecimal a2_inout)
                          { this.a2_inout = a2_inout; }
  public java.lang.String getA4Out()   { return a4_out; }
}
```

The `ToplevelGUser_g_Out` return type encapsulates the values of the `OUT` and `IN OUT` parameters to be passed back to the caller of the function. As in the preceding section, `oracle.sql.CLOB` is mapped to `String` by the `webservices10` style file.

Translation of Overloaded Methods

PL/SQL, like Java, lets you create overloaded methods, meaning two or more methods with the same name but different signatures. However, overloaded methods with different signatures in PL/SQL may have identical signatures in Java, especially in user subclasses. As an example, consider the following PL/SQL stored procedures:

```
PROCEDURE foo(x SYS.XMLTYPE);
PROCEDURE foo(x CLOB);
PROCEDURE foo(x NCHAR);
```

If you process these with a JPublisher setting of `-style=webservices-common`, then they will all have the same signature in Java:

```
void foo(String x);
void foo(String x);
void foo(String x);
```

See Also: "Style File Specifications and Locations" on page 3-24

JPublisher solves such naming conflicts by appending the first letter of the return type and the first letter of each argument type, as applicable, to the method name. If conflicts still remain, then a number is also appended. JPublisher solves the preceding conflict as follows:

```
void foo(String x);
void fooS(String x);
void fooS1(String x);
```

Note that PL/SQL does *not* allow overloading for types from the same family. The following, for example, is illegal:

```
PROCEDURE foo(x DECIMAL);
PROCEDURE foo(x INT);
PROCEDURE foo(x INTEGER);
```

Now, consider the procedures as functions instead, with return types from the same family. The following example is allowed because the argument types are different:

```
FUNCTION foo(x FLOAT) RETURN DECIMAL;
FUNCTION foo(x VARCHAR2) RETURN INT;
FUNCTION foo(x Student_T) RETURN INTEGER;
```

By default, these are mapped to Java methods as follows:

```
java.math.BigDecimal foo(Float x);
java.math.BigDecimal foo(String x);
java.math.BigDecimal foo(StudentT x);
```

JPublisher allows them all to be named `foo()` because now the signatures differ. However, if you want all method names to be unique, as is required for Web services, use the `unique` setting of the JPublisher `-methods` option. With `-methods=unique`, JPublisher publishes the methods as follows, using the naming mechanism described earlier:

```
java.math.BigDecimal foo(Float x);
java.math.BigDecimal fooBS(String x);
java.math.BigDecimal fooBS1(StudentT x);
```

> **See Also:** "Generation of Package Classes and Wrapper Methods" on page 6-32

Generation of SQLJ Classes

For the `-methods=all` setting, which is the default, or the `-methods=true` setting, JPublisher typically generates SQLJ classes for PL/SQL packages and object types, using both `ORAData` and `SQLData` implementations. An exception is that a SQLJ class is not generated if an object type does not define any methods, in which case the generated Java class does not require the SQLJ run time.

SQLJ classes include wrapper methods that invoke the server methods, or stored procedures, of object types and packages. This section describes how to use these classes.

> **See Also:** "JPublisher Usage of the SQLJ Implementation" on page 1-4 and "Backward Compatibility Option" on page 6-46

This section covers the following topics:

- Important Notes About Generation of SQLJ Classes
- Use of SQLJ Classes for PL/SQL Packages
- Use of SQLJ Classes for Object Types
- Connection Contexts and Instances in SQLJ Classes
- The setFrom(), setValueFrom(), and setContextFrom() Methods

Important Notes About Generation of SQLJ Classes

Note the following for JPublisher-generated SQLJ classes:

- If you are generating Java wrapper classes for a SQL type hierarchy and any of the types contains stored procedures, then by default, JPublisher generates SQLJ classes for all the SQL types and not just the types that have stored procedures.

> **Note:** You have the option of explicitly suppressing the generation of SQLJ classes through the JPublisher `-methods=false` setting. This results in all non-SQLJ classes.

- Classes produced by JPublisher include a `release()` method. If an instance of a JPublisher-generated wrapper class implicitly constructs a `DefaultContext` instance, then you should use the `release()` method to release this connection context instance when it is no longer needed. However, you can avoid this scenario by adhering to at least one of the following suggestions in creating and using the wrapper class instance:
  - Construct the wrapper class instance with an explicitly provided SQLJ connection context.
  - Associate the wrapper class instance explicitly with a SQLJ connection context instance through the `setConnectionContext()` method.
  - Use the static SQLJ default connection context instance implicitly for the wrapper class instance. This occurs if you do not supply any connection information.

    See Also: "Connection Contexts and Instances in SQLJ Classes" on page 4-10

- In the Oracle8*i* compatibility mode, instead of the constructor taking a `DefaultContext` instance or an instance of a user-specified class, there is a constructor that simply takes a `ConnectionContext` instance. This could be an instance of any class that implements the standard `sqlj.runtime.ConnectionContext` interface, including the `DefaultContext` class.

Use of SQLJ Classes for PL/SQL Packages

Take the following steps to use a class that JPublisher generates for a PL/SQL package:

1. Construct an instance of the class.
2. Call the wrapper methods of the class.

The constructors for the class associate a database connection with an instance of the class. One constructor takes a SQLJ `DefaultContext` instance or an instance of a class specified through the `-context` option when you run JPublisher. Another constructor takes a JDBC `Connection` instance. One constructor has no arguments. Calling the no-argument constructor is equivalent to passing the SQLJ default context to the constructor that takes a `DefaultContext` instance. JPublisher provides the constructor that takes a `Connection` instance for the convenience of JDBC programmers unfamiliar with SQLJ concepts, such as connection contexts and the default context.

> **See Also:** "Important Notes About Generation of SQLJ Classes" on page 4-7

The wrapper methods are all instance methods, because the connection context in the `this` object is used in the wrapper methods.

Because a class generated for a PL/SQL package has no instance data other than the connection context, you typically construct one class instance for each connection

context that you use. If the default context is the only one you use, then you can call the no-argument constructor once.

An instance of a class generated for a PL/SQL package does not contain copies of the PL/SQL package variables. It is not an `ORAData` class or a `SQLData` class, and you cannot use it as a host variable.

Use of SQLJ Classes for Object Types

To use an instance of a Java class that JPublisher generates for a SQL object type or a SQL `OPAQUE` type, you must first initialize the Java object. You can accomplish this in one of the following ways:

- Assign an already initialized Java object to your Java object.
- Retrieve a copy of a SQL object into your Java object. You can do this by using the SQL object as an `OUT` argument or as the function return of a JPublisher-generated wrapper method. You can also do this by retrieving the SQL object through JDBC calls that you write. If you are in a backward-compatibility mode and use SQLJ source files directly, then you can retrieve a copy of a SQL object through the SQLJ `#sql` statements.
- Construct the Java object with the no-argument constructor and set its attributes by using the `set`*`XXX`*`()` methods, or construct the Java object with the constructor that accepts values for all the object attributes. Subsequently, you need to use the `setConnection()` or `setConnectionContext()` method to associate the object with a database connection before calling any of its wrapper methods. If you do not explicitly associate the object with a JDBC or SQLJ connection before calling a method on it, then it becomes implicitly associated with the SQLJ default context.

  Other constructors for the class associate a connection with the class instance. One constructor takes a `DefaultContext` instance or an instance of a class specified through the `-context` option when you run JPublisher, and one constructor takes a `Connection` instance. The constructor that takes a `Connection` instance is provided for the convenience of JDBC programmers unfamiliar with SQLJ concepts, such as connection contexts and the default context.

  See Also: "Important Notes About Generation of SQLJ Classes" on page 4-7

Once you have initialized your Java object, you can do the following:

- Call the accessor methods of the object.
- Call the wrapper methods of the object.
- Pass the object to other wrapper methods.
- Use the object as a host variable in JDBC calls. If you are in a backward-compatibility mode and use SQLJ source files directly, then you can use the object as a host variable in the SQLJ `#sql` statements.

There is a Java attribute for each attribute of the corresponding SQL object type, with the `get`*`XXX`*`()` and `set`*`XXX`*`()` accessor methods for each attribute. JPublisher does not generate fields for the attributes. For example, for an attribute called `foo`, there is a corresponding Java attribute called `foo` and the accessor methods, `getFoo()` and `setFoo()`.

By default, the class includes wrapper methods that call the associated Oracle object methods, which reside and run on the server. Irrespective of what the server methods

are, the wrapper methods are all instance methods. The `DefaultContext` in the `this` object is used in the wrapper methods.

With Oracle mapping, JPublisher generates the following methods for the Oracle JDBC driver to use:

- `create()`
- `toDatum()`

These methods are specified in the `ORAData` and `ORADataFactory` interfaces and are generally not intended for your direct use. In addition, JPublisher generates the `setFrom(`*`otherObject`*`)`, `setValueFrom(`*`otherObject`*`)`, and `setContextFrom(`*`otherObject`*`)` methods that you can use to copy values or connection information from one object instance to another.

Connection Contexts and Instances in SQLJ Classes

The class that JPublisher uses in creating SQLJ connection context instances depends on how you set the `-context` option when you run JPublisher. The following classes can be used:

- A setting of `-context=DefaultContext`, which is the default setting, results in JPublisher using instances of the standard `sqlj.runtime.ref.DefaultContext` class.
- A setting of a user-defined class that is in `CLASSPATH` and implements the standard `sqlj.runtime.ConnectionContext` interface results in JPublisher using instances of that class.
- A setting of `-context=generated` results in the declaration of the static `_Ctx` connection context class in the JPublisher-generated class. JPublisher uses instances of this class for connection context instances. This is appropriate for the Oracle8*i* compatibility mode, but generally not recommended.

  See Also: "SQLJ Connection Context Classes" on page 6-20

  Note: It is no longer a routine, as it was in Oracle8*i* Database, for JPublisher to declare a `_ctx` connection context instance. However, this is used in the Oracle8*i* compatibility mode, with `_ctx` being declared as a `protected` instance of the static `_Ctx` connection context class.

  Unless you have legacy code that depends on `_ctx`, it is preferable to use the `getConnectionContext()` and `setConnectionContext()` methods to retrieve and manipulate connection context instances in JPublisher-generated classes.

Consider the following points in using SQLJ connection context instances or JDBC connection instances in instances of JPublisher-generated wrapper classes:

- Wrapper classes generated by JPublisher provide a `setConnectionContext()` method that you can use to explicitly specify a SQLJ connection context instance. The method is defined as follows:

  ```
  void setConnectionContext(conn_ctxt_instance);
  ```

  This installs the passed connection context instance as the SQLJ connection context in the wrapper class instance. The connection context instance must be an instance

of the class specified through the `-context` setting for JPublisher connection contexts, typically `DefaultContext`.

Note that the underlying JDBC connection must be compatible with the connection used to materialize the database object in the first place. Specifically, some objects may have attributes that are valid only for a particular connection, such as object reference types or BLOBs.

If you have already specified a connection context instance through the constructor, then you need not set it again using the `setConnectionContext()` method.

> **Note:** Using the `setConnectionContext()` method to explicitly set a connection context instance avoids the problem of the connection context not being closed properly. This problem occurs only with implicitly created connection context instances.

- Use either of the following methods of a wrapper class instance, as appropriate, to retrieve a connection or connection context instance:
  - `Connection getConnection()`
  - *`ConnCtxtType`* `getConnectionContext()`

  The `getConnectionContext()` method returns an instance of the connection context class specified through the JPublisher `-context` setting, typically `DefaultContext`.

  The returned connection context instance may be either an explicitly set instance or one that was created implicitly by JPublisher.

  > **Note:** These methods are available only in the generated SQLJ classes. If necessary, you can use the setting `-methods=always` to ensure that SQLJ classes are produced.

- If no connection context instance is explicitly set for a JPublisher-generated SQLJ class, then one will be created implicitly from the JDBC connection instance when the `getConnectionContext()` method is called.

  In this circumstance, at the end of processing, you need to use the `release()` method to free resources in the SQLJ run time. This prevents a possible memory leak.

The setFrom(), setValueFrom(), and setContextFrom() Methods

JPublisher provides the following utility methods in the generated SQLJ classes:

- `setFrom(`*`anotherObject`*`)`

  This method initializes the calling object from another object of the same base type, including connection and connection context information. An existing, implicitly created connection context object on the calling object is freed.

- `setValueFrom(`*`anotherObject`*`)`

  This method initializes the underlying field values of the calling object from another object of the same base type. This method does not transfer connection or connection context information.

- `setContextFrom(`*`anotherObject`*`)`

  This method initializes the connection and connection context information on the calling object from the connection setting of another object of the same base type. An existing, implicitly created, connection context object on the calling object is freed. This method does not transfer any information related to the object value.

Note that there is semantic equivalence between the `setFrom()` method and the combination of the `setValueFrom()` and `setContextFrom()` methods.

Generation of Non-SQLJ Classes

For a `-methods=false` setting, or when SQL object types do not define any methods, JPublisher does not generate wrapper methods for object types. In this case, the generated class does not require the SQLJ run time during execution. Therefore, JPublisher generates non-SQLJ classes, meaning classes that do not call the SQLJ run time application programming interfaces (APIs). All this is true regardless of whether you use an `ORAData` implementation or an `SQLData` implementation.

Notes:

- For the `-methods=false` setting, JPublisher does not generate code for PL/SQL packages, because they are not useful without wrapper methods.
- JPublisher generates the same Java code for reference, `VARRAY`, and nested table types regardless of whether the `-methods` option is set to `false` or `true`.

To use an instance of a class that JPublisher generates for an object type with the `-methods=false` setting or for a reference, `VARRAY`, or nested table type, you must first initialize the object.

You can initialize your object in one of the following ways:

- Assign an already initialized Java object to your Java object.
- Retrieve a copy of a SQL object into your Java object. You can do this by using the SQL object as an `OUT` argument or as the function return accessed through a JPublisher-generated wrapper method in some other class. You can also do this by retrieving the SQL object through JDBC calls that you write. If you are in a backward-compatibility mode and using SQLJ source files directly, then you can retrieve a copy of a SQL object through the SQLJ `#sql` statements.
- Construct the Java object with a no-argument constructor and initialize its data, or construct the Java object based on its attribute values.

Unlike the constructors generated in SQLJ classes, the constructors generated in non-SQLJ classes do not take a connection argument. Instead, when your object is passed to or returned from a JDBC `Statement`, `CallableStatement`, or `PreparedStatement` object, JPublisher applies the connection it uses to construct the `Statement`, `CallableStatement`, or `PreparedStatement` object.

This does not mean you can use the same object with different connections at different times, which is not always possible. An object may have a subcomponent that is valid only for a particular connection, such as a reference or a `BLOB`.

To initialize the object data, use the `set`*`XXX`*`()` methods, if your class represents an object type, or the `setArray()` or `setElement()` method, if your class represents a

VARRAY or nested table type. If your class represents a reference type, then you can construct only a null reference. All non-null references come from the database.

Once you have initialized your object, you can do the following:

- Pass the object to wrapper methods in other classes.
- Use the object as a host variable in JDBC calls. If you are in a backward-compatibility mode and use SQLJ source files directly, then you can use the object in the SQLJ `#sql` statements.
- Call the methods that read and write the state of the object. These methods operate on the Java object in your program and do not affect data in the database. You can read and write the state of the object in the following ways:
  - For a class that represents an object type, call the `get`*`XXX`*`()` and `set`*`XXX`*`()` accessor methods.
  - For a class that represents a `VARRAY` or nested table, call the `getArray()`, `setArray()`, `getElement()`, and `setElement()` methods.

    The `getArray()` and `setArray()` methods return or modify an array as a whole. The `getElement()` and `setElement()` methods return or modify individual elements of the array.

    If you want to update the data in the database, then you need to re-insert the Java array into the database.
- You cannot modify an object reference, because it is an immutable entity. However, you can read and write the SQL object that it references by using the `getValue()` and `setValue()` methods.

  The `getValue()` method returns a copy of the SQL object that is being referenced by the object reference. The `setValue()` method updates a SQL object type instance in the database by taking an instance of the Java class that represents the object type as input. Unlike the `get`*`XXX`*`()` and `set`*`XXX`*`()` accessor methods of a class generated for an object type, the `getValue()` and `setValue()` methods read and write SQL objects.

  Note that both `getValue()` and `setValue()` result in a database round trip to read or write the value of the underlying database object that the reference points to.

You can use the `getORADataFactory()` method in the JDBC code to return an `ORADataFactory` object. You can pass this `ORADataFactory` object to the `getORAData()` method in the `ArrayDataResultSet`, `OracleCallableStatement`, and `OracleResultSet` classes in the `oracle.jdbc` package. The Oracle JDBC driver uses the `ORADataFactory` object to create instances of your JPublisher-generated class.

In addition, classes representing `VARRAY` and nested table types have methods that implement features of the `oracle.sql.ARRAY` class. These methods are:

- `getBaseTypeName()`
- `getBaseType()`
- `getDescriptor()`

However, JPublisher-generated classes for `VARRAY` and nested table types do not extend the `oracle.sql.ARRAY` class.

With Oracle mapping, JPublisher generates the following methods for the Oracle JDBC driver to use:

- `create()`
- `toDatum()`

These methods are specified in the `ORAData` and `ORADataFactory` interfaces and are not generally intended for direct use. However, you may want to use them if converting from one object reference Java wrapper type to another.

Generation of Java Interfaces

JPublisher has the ability to generate interfaces as well as classes. This feature is especially useful for Web services, because it eliminates the necessity to manually create Java interfaces that represent the API from which WSDL content is generated.

The `-sql` option supports the following syntax:

```
-sql=sql_package_or_type:JavaClass#JavaInterface
```

or:

```
-sql=sql_package_or_type:JavaClass:JavaUserSubclass#JavaSubInterface
```

Whenever an interface name is specified in conjunction with a class, then the public attributes or wrapper methods or both of that class are provided in the interface, and the generated class implements the interface.

> **See Also:** "Publishing User-Defined SQL Types" on page 2-1 and "Publishing PL/SQL Packages" on page 2-4

You can specify an interface for either the generated class or the user subclass, but not both. The difference between an interface for a generated base class and one for a user subclass involves Java-to-Java type transformations. Method signatures in the subclass may be different from signatures in the base class because of Java-to-Java mappings.

> **See Also:** "JPublisher Styles and Style Files" on page 3-23

JPublisher Subclasses

In translating a SQL user-defined type, you may want to enhance the functionality of the custom Java class generated by JPublisher.

One way to accomplish this is to manually add methods to the class generated by JPublisher. However, this is not advisable if you anticipate running JPublisher in the future to regenerate the class. If you regenerate a class that you have modified in this way, then your changes, such as the methods you have added, will be overwritten. Even if you direct JPublisher output to a separate file, you still must merge your changes into the file.

The preferred way to enhance the functionality of a generated class is to extend the class. JPublisher has a mechanism for this, where it will generate the original base class along with a stub subclass, which you can customize as desired. Wherever the SQL type is referenced in code, such as when it is used as an argument, the SQL type will be mapped to the subclass rather than to the base class.

There is also a scenario for JPublisher-generated subclasses for Java-to-Java type transformations. You may have situations in which JPublisher mappings from SQL types to Java types use Java types unsuitable for your purposes; for example, types unsupported by Web services. JPublisher uses a mechanism of styles and style files to allow an additional Java-to-Java transformation step, to use a Java type that is suitable.

The following topics are covered in this section:

- Extending JPublisher-Generated Classes
- JPublisher-Generated Subclasses for Java-to-Java Type Transformations

Extending JPublisher-Generated Classes

Suppose you want JPublisher to generate the `JAddress` class from the `ADDRESS` SQL object type. You also want to write a class, `MyAddress`, to represent `ADDRESS` objects, where `MyAddress` extends the functionality that `JAddress` provides.

Under this scenario, you can use JPublisher to generate both a base Java class, `JAddress`, and an initial version of a subclass, `MyAddress`, to which you can add the desired functionality. You then use JPublisher to map `ADDRESS` objects to the `MyAddress` class instead of the `JAddress` class.

To do this, JPublisher alters the code it generates in the following ways:

- It generates the `MyAddressRef` reference class instead of `JAddressRef`.
- It uses the `MyAddress` class, instead of the `JAddress` class, to represent attributes with the SQL type `ADDRESS` or to represent `VARRAY` and nested table elements with the SQL type `ADDRESS`.
- It uses the `MyAddress` factory, instead of the `JAddress` factory, when the `ORADataFactory` interface is used to construct Java objects with the SQL type `ADDRESS`.
- It generates or regenerates the code for the `JAddress` class. In addition, it generates an initial version of the code for the `MyAddress` class, which you can then modify to insert your own additional functionality. However, if the source file for the `MyAddress` class already exists, then it is left untouched by JPublisher.

> **See Also:** "Changes in JPublisher Behavior Between Oracle8i Database and Oracle9i Database" on page 5-7

Syntax for Mapping to Alternative Classes

JPublisher has the functionality to streamline the process of mapping to alternative classes. Use the following syntax in your `-sql` command-line option setting:

```
-sql=object_type:generated_base_class:map_class
```

For the `MyAddress/JAddress` example, it is:

```
-sql=ADDRESS:JAddress:MyAddress
```

If you were to enter the line in the `INPUT` file instead of on the command line, it would look like this:

```
SQL ADDRESS GENERATE JAddress AS MyAddress
```

In this syntax, `JAddress` is the name of the base class that JPublisher generates, in `JAddress.java`, but `MyAddress` is the name of the class that actually maps to `ADDRESS`. You are ultimately responsible for the code in `MyAddress.java`. Update this as necessary to add your custom functionality. If you retrieve an object that has an `ADDRESS` attribute, then this attribute is created as an instance of `MyAddress`. Or, if you retrieve an `ADDRESS` object directly, then it is retrieved into an instance of `MyAddress`.

> **See Also:** "Declaration of Object Types and Packages to Translate" on page 6-14 and "INPUT File Structure and Syntax" on page 6-52

Format of the Class that Extends the Generated Class

For convenience, an initial version of the user subclass is automatically generated by JPublisher, unless it already exists. This subclass is where you place your custom code. For example, the `MyAddress.java` file generated by JPublisher in the preceding example.

Note the following:

- The class has a no-argument constructor. The easiest way to construct a properly initialized object is to invoke the constructor of the superclass, either explicitly or implicitly.
- The class implements the `ORAData` interface or the `SQLData` interface. This happens implicitly by inheriting the necessary methods from the superclass.
- When extending an `ORAData` class, the subclass also implements the `ORADataFactory` interface, with an implementation of the `create()` method, as shown:

  ```
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
     return create(new UserClass(),d,sqlType);
  }
  ```

  However, when the class is part of an inheritance hierarchy, the generated method changes to `protected ORAData createExact()`, with the same signature and body as `create()`.

JPublisher-Generated Subclasses for Java-to-Java Type Transformations

JPublisher style files enable you to specify Java-to-Java type mappings. A typical use for such mappings is to ensure that generated classes can be used in Web services. As a particular example, `CLOB` types, such as `java.sql.Clob` and `oracle.sql.CLOB`, cannot be used in Web services. However, the data can be used if converted to a type that is supported by Web services, such as `java.lang.String`.

If you use the JPublisher `-style` option to specify a style file, then JPublisher generates subclasses that implement the Java-to-Java type mappings specified in the style file. This includes the use of holder classes for handling output arguments and data corresponding to PL/SQL `OUT` or `IN OUT` types.

> **See Also:** "JPublisher Styles and Style Files" on page 3-23 and "Passing Output Parameters in JAX-RPC Holders" on page 4-4

For example, consider the following PL/SQL package, `foo_pack`, consisting of the stored function, `foo`:

```
CREATE OR REPLACE PACKAGE foo_pack AS
   FUNCTION foo(a IN OUT SYS.XMLTYPE, b INTEGER) RETURN CLOB;
END;
/
```

Assume that you translate the `foo_pack` package as follows:

```
% jpub -u scott/tiger -s foo_pack:FooPack -style=webservices10
```

This command uses the style file `webservices10.properties` for Java-to-Java type mappings. Among other things, the `webservices10.properties` file specifies the following:

- The mapping of the `oracle.sql.SimpleXMLType` Java type, which is not supported by Web services, to the `javax.xml.transform.Source` Java type. This is specified in the style file as follows:

  ```
  SOURCETYPE oracle.sql.SimpleXMLType
  TARGETTYPE javax.xml.transform.Source
  ...
  ```

- The use of holder classes for PL/SQL `OUT` and `IN OUT` arguments:

  ```
  jpub.outarguments=holder
  ```

  This setting directs JPublisher to use instances of the appropriate holder class, in this case `javax.xml.rpc.holders.SourceHolder`, for the PL/SQL output argument of the `XMLTYPE` type.

- The inclusion of `webservices-common.properties`:

  ```
  INCLUDE webservices-common
  ```

> **Note:** This style file is supplied by Oracle and is appropriate for using Web services in an Oracle Database 10*g* environment.

The `webservices-common.properties` file, which is also supplied by Oracle, specifies the following:

- The mapping of `SYS.XMLTYPE` to `oracle.sql.SimpleXMLType` in the JPublisher default type map:

  ```
  jpub.adddefaulttypemap=SYS.XMLTYPE:oracle.sql.SimpleXMLType
  ```

- A code generation naming pattern:

  ```
  jpub.genpattern=%2Base:%2User#%2
  ```

  Based on the `-s foo_pack:FooPack` specification to JPublisher, the `genpattern` setting results in the generation of the `FooPack` interface, the base class `FooPackBase`, and the user subclass `FooPackUser`, which extends `FooPackBase` and implements `FooPack`.

  > **See Also:** "Class and Interface Naming Pattern" on page 6-30

- The mapping of the `oracle.sql.CLOB` Java type, which is not supported by Web services, to the `java.lang.String` Java type:

  ```
  SOURCETYPE oracle.sql.CLOB
  TARGETTYPE java.lang.String
  ...
  ```

Recall the calling sequence for the stored function `foo`:

```
FUNCTION foo(a IN OUT SYS.XMLTYPE, b INTEGER) RETURN CLOB;
```

The base class generated by JPublisher, `FooPackBase`, has the following corresponding method declaration:

```
public oracle.sql.CLOB _foo (oracle.sql.SimpleXMLType a[], Integer b)
```

The base class uses an array to hold the output argument.

See Also: "Passing Output Parameters in Arrays" on page 4-2

The user subclass has the following corresponding method declaration:

```
public java.lang.String foo (SourceHolder _xa_inout_x, Integer b)
```

This declaration is because of the specified mapping of `oracle.sql.SimpleXMLType` to `javax.xml.transform.Source`, the specified use of holder classes for output arguments, and the specified mapping of `oracle.sql.CLOB` to `java.lang.String`.

The following is the code for the `SourceHolder` class, which is the holder class for `javax.xml.transform.Source`:

```
// Holder class for javax.xml.transform.Source
public class SourceHolder implements javax.xml.rpc.holders.Holder
{
   public javax.xml.transform.Source value;
   public SourceHolder() { }
   public SourceHolder(javax.xml.transform.Source value)
   { this.value = value; }
}
```

Generated user subclasses employ the following general functionality for Java-to-Java type transformations in the wrapper method:

```
User subclass method
{
     Enter Holder layer (extract IN data from the holder)
          Enter Java-to-Java mapping layer (from target to source)
               Call base class method (uses JDBC to invoke wrapped procedure)
          Exit Java-to-Java mapping layer (from source to target)
     Exit Holder layer (update the holder)
}
```

For this example, the `foo()` method of the `FooPackUser` class is defined as follows:

```
foo (SourceHolder, Integer)
{
     SourceHolder -> Source
          Source -> SimpleXMLType
               _foo (SimpleXMLType[], Integer);
          SimpleXMLType -> Source
     Source -> SourceHolder
}
```

Note: Do not confuse the term source with the class name `Source`. In this example, `Source` is a target type and `SimpleXMLType` is the corresponding source type.

The holder layer retrieves and assigns the holder instance.

In the example, the holder layer in `foo()` performs the following:

1. It retrieves a `Source` object from the `SourceHolder` object that is passed in to the `foo()` method.
2. After processing, which occurs inside the type conversion layer, it assigns the `SourceHolder` object from the `Source` object that was retrieved and processed.

The type conversion layer first takes the target type, `TARGETTYPE` from the style file, and converts it to the source type, `SOURCETYPE` from the style file. It then calls the corresponding method in the base class, which uses JDBC to invoke the wrapped stored function. Finally, it converts the source type returned by the base class method back into the target type to return to the holder layer.

In this example, the type conversion layer in `foo()` performs the following:

1. It takes the `Source` object from the holder layer.
2. It converts the `Source` object to a `SimpleXMLType` object.
3. It passes the `SimpleXMLType` object to the `_foo()` method of the base class, which uses JDBC to invoke the `foo` stored function.
4. It takes the `SimpleXMLType` object returned by the `_foo()` method.
5. It converts the `SimpleXMLType` object back to a `Source` object for the holder layer.

See Also: "Generated Code: User Subclass for Java-to-Java Transformations" on page A-1

Support for Inheritance

This section describes the inheritance support for the `ORAData` types and explains the following related topics:

- How JPublisher implements support for inheritance
- Why a reference class for a subtype does not extend the reference class for the base type, and how you can convert from one reference type to another reference type, typically a subclass or superclass

This section covers the following topics:

ORAData Object Types and Inheritance

Consider the following SQL object types:

```
CREATE TYPE PERSON AS OBJECT (
...
) NOT FINAL;

CREATE TYPE STUDENT UNDER PERSON (
...
);

CREATE TYPE INSTRUCTOR UNDER PERSON (
...
```

```
);
```

Consider the following JPublisher command to create corresponding Java classes:

```
% jpub -user=scott/tiger
       -sql=PERSON:Person,STUDENT:Student,INSTRUCTOR:Instructor -usertypes=oracle
```

In this example, JPublisher generates a `Person` class, a `Student` class, and an `Instructor` class. The `Student` and `Instructor` classes extend the `Person` class, because `STUDENT` and `INSTRUCTOR` are subtypes of `PERSON`.

The class at the root of the inheritance hierarchy, `Person` in this example, contains full information for the entire inheritance hierarchy and automatically initializes its type map with the required information. As long as you use JPublisher to generate all the required classes of a class hierarchy together, no additional action is required. The type map of the class hierarchy is appropriately populated.

This section covers the following topics:

- Precautions when Combining Partially Generated Type Hierarchies
- Mapping of Type Hierarchies in JPublisher-Generated Code

Precautions when Combining Partially Generated Type Hierarchies

If you run JPublisher several times on a SQL type hierarchy, each time generating only part of the corresponding Java wrapper classes, then you must take precautions in the user application to ensure that the type map at the root of the class hierarchy is properly initialized.

In our previous example, you may have run the following JPublisher commands:

```
% jpub -user=scott/tiger -sql=PERSON:Person,STUDENT:Student -usertypes=oracle
% jpub -user=scott/tiger -sql=PERSON:Person,INSTRUCTOR:Instructor
       -usertypes=oracle
```

In this case, you should create instances of the generated classes, at least of the leaf classes, before using these mapped types in your code. For example:

```
new Instructor(); // required
new Student();    // required
new Person();     // optional
```

Mapping of Type Hierarchies in JPublisher-Generated Code

The `Person` class includes the following method:

```
Person create(oracle.sql.Datum d, int sqlType)
```

This method converts a `Datum` instance to its representation as a custom Java object. It is called by the Oracle JDBC driver whenever a SQL object declared to be a `PERSON` is retrieved into a `Person` variable. The SQL object, however, may actually be a `STUDENT` object. In this case, the `create()` method must create a `Student` instance rather than a `Person` instance.

To handle this kind of situation, the `create()` method of a custom Java class must be able to create instances of any subclass that represents a subtype of the SQL object type corresponding to the `oracle.sql.Datum` argument. This ensures that the actual type of the created Java object matches the actual type of the SQL object. The custom Java class may or may not be created by JPublisher.

However, the code for the `create()` method in the root class of a custom Java class hierarchy need not mention the subclasses. In fact, if it *did* mention the subclasses, then you would have to modify the code for the base class whenever you write or create a new subclass. The base class is modified automatically if you use JPublisher to regenerate the entire class hierarchy. But regenerating the hierarchy may not always be possible. For example, you may not have access to the source code for the Java classes being extended.

Instead, code generated by JPublisher permits incremental extension of a class hierarchy by creating a static initialization block in each subclass of the custom Java class hierarchy. This static initialization block initializes a data structure declared in the root-level Java class, giving the root class the information it needs about the subclass. When an instance of a subclass is created at run time, the type is registered in the data structure. Because of this implicit mapping mechanism, no explicit type map, such as those required in the `SQLData` scenarios, is required.

Note: This implementation makes it possible to extend existing classes without having to modify them, but it also carries a penalty. The static initialization blocks of the subclasses must be processed before the class hierarchy can be used to read objects from the database. This occurs if you instantiate an object of each subclass by calling `new()`. It is sufficient to instantiate just the leaf classes, because the constructor for a subclass invokes the constructor for its immediate superclass.

As an alternative, you can generate or regenerate the entire class hierarchy, if it is feasible.

ORAData Reference Types and Inheritance

This section shows how to convert from one custom reference class to another and also explains why a custom reference class generated by JPublisher for a subtype does not extend the reference classes of the base type.

This section covers the following topics:

- Casting a Reference Type Instance into Another Reference Type
- Why Reference Type Inheritance Does Not Follow Object Type Inheritance
- Manually Converting Between Reference Types
- Example: Manually Converting Between Reference Types

Casting a Reference Type Instance into Another Reference Type

Revisiting the example in "ORAData Object Types and Inheritance" on page 4-19, `PersonRef`, `StudentRef`, and `InstructorRef` are obtained for strongly typed references, in addition to the underlying object type wrapper classes.

There may be situations in which you have a `StudentRef` instance, but you want to use it in a context that requires a `PersonRef` instance. In this case, use the static method, `cast()`, generated in strongly typed reference classes:

```
StudentRef s_ref = ...;
PersonRef p_ref = PersonRef.cast(s_ref);
```

Conversely, you may have a `PersonRef` instance and know that you can narrow it to an `InstructorRef` instance:

```
PersonRef pr = ...;
InstructorRef ir = InstructorRef.cast(pr);
```

Why Reference Type Inheritance Does Not Follow Object Type Inheritance

The example here helps explain why it is not desirable for reference types to follow the hierarchy of their related object types. Consider again a subset of the example given in the previous section (repeated here for convenience):

```
CREATE TYPE PERSON AS OBJECT (
...
) NOT FINAL;

CREATE TYPE STUDENT UNDER PERSON (
...
);
```

And consider the following JPublisher command:

```
% jpub -user=scott/tiger -sql=PERSON:Person,STUDENT:Student -usertypes=oracle
```

In addition to generating the `Person` and `Student` Java types, JPublisher generates `PersonRef` and `StudentRef` types.

Because the `Student` class extends the `Person` class, you may expect `StudentRef` to extend `PersonRef`. However, this is not the case, because the `StudentRef` class can provide more compile-time type safety as an independent class than as a subtype of `PersonRef`. Additionally, a `PersonRef` object can perform something that a `StudentRef` object cannot, such as modifying a `Person` object in the database.

The most important methods of the `PersonRef` class are the following:

- `Person getValue()`
- `void setValue(Person c)`

The corresponding methods of the `StudentRef` class are as follows:

- `Student getValue()`
- `void setValue(Student c)`

If the `StudentRef` class extended the `PersonRef` class, then the following problems would occur:

- Java would not permit the `getValue()` method in `StudentRef` to return a `Student` object when the method it overrides in the `PersonRef` class returns a `Person` object, even though this is arguably a sensible thing to do.
- The `setValue()` method in `StudentRef` would not override the `setValue()` method in `PersonRef`, because the two methods have different signatures.

You cannot remedy these problems by giving the `StudentRef` methods the same signatures and result types as the `PersonRef` methods, because the additional type safety provided by declaring an object as a `StudentRef`, rather than as a `PersonRef`, would be lost.

Manually Converting Between Reference Types

You cannot convert one reference type to another directly, because reference types do not follow the hierarchy of their related object types. This is a limitation of JPublisher. For background information, this section explains how the generated `cast()` methods work to convert from one reference type to another.

> **Note:** It is *not* recommended that you follow these manual steps. They are presented here for illustration only. You can use the `cast()` method instead.

The following example outlines the code that could be used to convert from the `XxxxRef` reference type to the `YyyyRef` reference type:

```
java.sql.Connection conn = ...;  // get underlying JDBC connection
XxxxRef xref = ...;
YyyyRef yref = (YyyyRef) YyyyRef.getORADataFactory().
               create(xref.toDatum(conn),oracle.jdbc.OracleTypes.REF);
```

This conversion consists of two steps, each of which can be useful in its own right.

1. Convert `xref` from its strong `XxxxRef` type to the weak `oracle.sql.REF` type:

   ```
   oracle.sql.REF ref  = (oracle.sql.REF) xref.toDatum(conn);
   ```

2. Convert from the `oracle.sql.REF` type to the target `YyyyRef` type:

   ```
   YyyyRef yref = (YyyyRef) YyyyRef.getORADataFactory().
                            create(ref,oracle.jdbc.OracleTypes.REF);
   ```

> **Note:** This conversion does not include any type-checking. Whether this conversion is actually permitted depends on your application and on the SQL schema you are using.

Example: Manually Converting Between Reference Types

The following example, including the SQL definitions and Java code, illustrates the points of the preceding discussion.

SQL Definitions Consider the following SQL definitions:

```
CREATE TYPE person_t AS OBJECT (ssn NUMBER, name VARCHAR2(30), dob DATE) NOT
FINAL;
/
SHOW ERRORS

CREATE TYPE instructor_t UNDER person_t (title VARCHAR2(20)) NOT FINAL;
/
SHOW ERRORS

CREATE TYPE instructorPartTime_t UNDER instructor_t (num_hours NUMBER);
/
SHOW ERRORS

CREATE TYPE student_t UNDER person_t (deptid NUMBER, major VARCHAR2(30)) NOT
FINAL;
/
SHOW ERRORS

CREATE TYPE graduate_t UNDER student_t (advisor instructor_t);
/
SHOW ERRORS

CREATE TYPE studentPartTime_t UNDER student_t (num_hours NUMBER);
```

```
/
SHOW ERRORS

CREATE TABLE person_tab OF person_t;

INSERT INTO person_tab VALUES (1001, 'Larry', TO_DATE('11-SEP-60'));

INSERT INTO person_tab VALUES (instructor_t(1101, 'Smith', TO_DATE('09-OCT-1940'),
'Professor'));

INSERT INTO person_tab VALUES (instructorPartTime_t(1111, 'Myers',
TO_DATE('10-OCT-65'), 'Adjunct Professor', 20));

INSERT INTO person_tab VALUES (student_t(1201, 'John', To_DATE('01-OCT-78'), 11,
'EE'));

INSERT INTO person_tab VALUES (graduate_t(1211, 'Lisa', TO_DATE('10-OCT-75'), 12,
'ICS', instructor_t(1101, 'Smith', TO_DATE ('09-OCT-40'), 'Professor')));

INSERT INTO person_tab VALUES (studentPartTime_t(1221, 'Dave',
TO_DATE('11-OCT-70'), 13, 'MATH', 20));
```

JPublisher Mappings Assume the following mappings when you run JPublisher:

```
Person_t:Person,instructor_t:Instructor,instructorPartTime_t:InstructorPartTime,
graduate_t:Graduate,studentPartTime_t:StudentPartTime
```

SQLJ Class Here is a SQLJ class with an example of reference type conversion:

```
import java.sql.*;
import oracle.jdbc.*;
import oracle.sql.*;

public class Inheritance
{
  public static void main(String[] args) throws SQLException
  {
    System.out.println("Connecting.");
    java.sql.DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
    oracle.jdbc.OracleConnection conn =
              (oracle.jdbc.OracleConnection) java.sql.DriverManager.getConnection
              ("jdbc:oracle:oci8:@", "scott", "tiger");
    // The following is only required in 9.0.1
    // or if the Java class hierarchy was created piecemeal
    System.out.println("Initializing type system.");
    new Person();
    new Instructor();
    new InstructorPartTime();
    new StudentT();
    new StudentPartTime();
    new Graduate();
    PersonRef p_ref;
    InstructorRef i_ref;
    InstructorPartTimeRef ipt_ref;
    StudentTRef s_ref;
    StudentPartTimeRef spt_ref;
    GraduateRef g_ref;
    OraclePreparedStatement stmt =
                (OraclePreparedStatement)conn.prepareStatement
                ("select ref(p) FROM PERSON_TAB p WHERE p.NAME=:1");
```

```
OracleResultSet rs;

System.out.println("Selecting a person.");
stmt.setString(1, "Larry");
rs = (OracleResultSet) stmt.executeQuery();
rs.next();
p_ref = (PersonRef) rs.getORAData(1, PersonRef.getORADataFactory());
rs.close();

System.out.println("Selecting an instructor.");
stmt.setString(1, "Smith");
rs = (OracleResultSet) stmt.executeQuery();
rs.next();
i_ref = (InstructorRef) rs.getORAData(1, InstructorRef.getORADataFactory());
rs.close();

System.out.println("Selecting a part time instructor.");
stmt.setString(1, "Myers");
rs = (OracleResultSet) stmt.executeQuery();
rs.next();
ipt_ref = (InstructorPartTimeRef) rs.getORAData
          (1, InstructorPartTimeRef.getORADataFactory());
rs.close();

System.out.println("Selecting a student.");
stmt.setString(1, "John");
rs = (OracleResultSet) stmt.executeQuery();
rs.next();
s_ref = (StudentTRef) rs.getORAData(1, StudentTRef.getORADataFactory());
rs.close();

System.out.println("Selecting a part time student.");
stmt.setString(1, "Dave");
rs = (OracleResultSet) stmt.executeQuery();
rs.next();
spt_ref = (StudentPartTimeRef) rs.getORAData
          (1, StudentPartTimeRef.getORADataFactory());
rs.close();

System.out.println("Selecting a graduate student.");
stmt.setString(1, "Lisa");
rs = (OracleResultSet) stmt.executeQuery();
rs.next();
g_ref = (GraduateRef) rs.getORAData(1, GraduateRef.getORADataFactory());
rs.close();
stmt.close();

// Assigning a part-time instructor ref to a person ref
System.out.println("Assigning a part-time instructor ref to a person ref");
oracle.sql.Datum ref = ipt_ref.toDatum(conn);
PersonRef pref = (PersonRef) PersonRef.getORADataFactory().
                                        create(ref,OracleTypes.REF);
// or just use: PersonRef pref = PersonRef.cast(ipt_ref);
// Assigning a person ref to an instructor ref
System.out.println("Assigning a person ref to an instructor ref");
InstructorRef iref = (InstructorRef) InstructorRef.getORADataFactory().
                                  create(pref.toDatum(conn), OracleTypes.REF);
// or just use: InstructorRef iref = InstructorRef.cast(pref);
// Assigning a graduate ref to an part time instructor ref.
// This should produce an error, demonstrating that refs
```

```
    // are type safe.
    System.out.println ("Assigning a graduate ref to a part time instructor ref");
    InstructorPartTimeRef iptref =
                (InstructorPartTimeRef) InstructorPartTimeRef.getORADataFactory().
                create(g_ref.toDatum(conn), OracleTypes.REF);
    // or just use: InstructorPartTimeRef iptref =
    // InstructorPartTimeRef.cast(g_ref);
    conn.close();
  }
}
```

SQLData Object Types and Inheritance

If you use the JPublisher `-usertypes=jdbc` setting instead of `-usertypes=oracle`, then the custom Java class generated by JPublisher implements the standard `SQLData` interface instead of the Oracle `ORAData` interface. The standard `SQLData` methods, `readSQL()` and `writeSQL()`, provide functionality equivalent to the `ORAData/ORADataFactory` methods, `create()` and `toDatum()`, for reading and writing data.

When JPublisher generates `SQLData` classes corresponding to a SQL hierarchy, the Java types follow the same hierarchy as the SQL types. This is similar to the case when JPublisher generates `ORAData` classes corresponding to a hierarchy of SQL object types. However, `SQLData` implementations do not offer the implicit mapping intelligence that JPublisher automatically generates in `ORAData` classes.

In a `SQLData` scenario, you must manually provide a type map to ensure correct mapping between SQL object types and Java types. In a JDBC application, you can properly initialize the default type map for your connection or you can explicitly provide a type map as a `getObject()` input parameter.

See Also: *Oracle Database JDBC Developer's Guide and Reference*

In addition, note that there is no support for strongly typed object references in an `SQLData` implementation. All object references are weakly typed `java.sql.Ref` instances.

Effects of Using SQL FINAL, NOT FINAL, NOT INSTANTIABLE

This section discusses the effect of using the SQL modifiers `FINAL`, `NOT FINAL`, or `NOT INSTANTIABLE` on JPublisher-generated wrapper classes.

Using the SQL modifier `FINAL` or `NOT FINAL` on a SQL type or on a method of a SQL type has no effect on the generated Java wrapper code. This ensures that, in all cases, JPublisher users are able to customize generated Java wrapper classes by extending the classes and overriding the generated behavior.

Using the `NOT INSTANTIABLE` SQL modifier on a method of a SQL type results in no code being generated for that method in the Java wrapper class. Therefore, to call such a method, you must cast to some wrapper class that corresponds to an instantiable SQL subtype.

Using `NOT INSTANTIABLE` on a SQL type results in the corresponding wrapper class being generated with `protected` constructors. This will remind you that instances of that class can be created only through subclasses that correspond to the instantiable SQL types.

5

Additional Features and Considerations

This chapter covers additional features and considerations for your use of JPublisher:

- Summary of JPublisher Support for Web Services
- Features to Filter JPublisher Output
- Backward Compatibility and Migration

Summary of JPublisher Support for Web Services

The following sections summarize key JPublisher features for Web services. Most features relate to Web services call-ins to the database, covering JPublisher features that make SQL, PL/SQL, and server-side Java classes accessible to Web services clients. There are also features and options to support Web services call-outs from the database.

- Summary of Support for Web Services Call-Ins to the Database
- Support for Web Services Call-Outs from the Database
- Server-Side Java Invocation (Call-in)

See Also:

- *Oracle Database Java Developer's Guide* for additional information about Oracle Database Web services
- *Oracle Application Server Web Services Developer's Guide* for general information about Oracle features for Web services

Summary of Support for Web Services Call-Ins to the Database

The following JPublisher features support Web services call-ins to code running in the Oracle Database.

- Generation of Java interfaces

  By using extended functionality of the `-sql` option, JPublisher can generate Java interfaces. This functionality eliminates the necessity to manually generate Java interfaces that represent the application programming interface (API) from which Web Services Description Language (WSDL) content is to be generated. Prior to Oracle Database 10*g*, JPublisher could generate classes but not interfaces.

  See Also: "Generation of Java Interfaces" on page 4-14

- JPublisher styles and style files

Style files, along with the related `-style` option, enable Java-to-Java type mappings that ensure that generated classes can be used in Web services. In particular, Oracle provides the following style files to support Web services:

```
/oracle/jpub/mesg/webservices-common.properties
/oracle/jpub/mesg/webservices10.properties
/oracle/jpub/mesg/webservices9.properties
```

See Also: "JPublisher Styles and Style Files" on page 3-23

- `REF CURSOR` returning and result set mapping

  The `java.sql.ResultSet` type is not supported by Web services, which affects stored procedures and functions that return `REF CURSOR` types. JPublisher supports alternative mappings that allow the use of query results with Web services.

  See Also: "REF CURSOR Types and Result Sets Mapping" on page 3-7

- Options to filter what JPublisher publishes

  There are several features for specifying or filtering JPublisher output, particularly to ensure that JPublisher-generated code can be exposed as Web services. By using the extended functionality of the `-sql` option, you can publish a specific subset of stored procedures. Using the `-filtertypes` and `-filtermodes` options, you can publish stored procedures based on the modes or types of parameters or return values. Using the `-generatebean` option, you can specify that generated methods satisfy the JavaBeans specification.

  See Also: "Features to Filter JPublisher Output" on page 5-4

- Support for calling Java classes in the database

  JPublisher uses the native Java interface for calls directly from a client-side Java stub, generated by JPublisher through the `-java` option, to the server-side Java code. Prior to Oracle Database 10*g*, server-side Java code could be called only through a PL/SQL wrapper that had to be created manually. This PL/SQL wrapper was also known as a call spec. In Oracle Database 10*g* release 2 (10.2), Web services call-ins of Java classes are supported in two modes, dynamic invocation mode and PL/SQL wrapper mode.

  See Also: "Publishing Server-Side Java Classes Through Native Java Interface" on page 2-11 and "Server-Side Java Invocation (Call-in)" on page 5-3

- Support for publishing SQL queries or DML statements

  JPublisher provides the `-sqlstatement` option to take a particular `SELECT`, `UPDATE`, `INSERT`, or `DELETE` statement and publish it as a method on a Java class that can be published as a Web service.

- Support for unique method names

  To meet Web services requirements, you can instruct JPublisher to disallow overloaded methods and always use unique method names instead.

See Also: "Generation of Package Classes and Wrapper Methods" on page 6-32

Support for Web Services Call-Outs from the Database

JPublisher supports Web services call-outs from the Oracle Database. The Web services client code is written in SQL, PL/SQL, or Java and it runs on the database and invokes Web services elsewhere. This support is provided through the `-proxywsdl` and `-httpproxy` options. In addition, the `-proxyopts` and `-proxyclasses` options may possibly be relevant, but typically do not require any special settings for Web services.

Here is a summary of the key options:

- `-proxywsdl=`*`URL`*

  Use this option to generate Web services client proxy classes, given the WSDL document at the specified URL. This option also generates additional wrapper classes to expose instance methods as static methods and generates PL/SQL wrappers.

- `-httpproxy=`*`proxy_URL`*

  Where a WSDL document is accessed through a firewall, use this option to specify a proxy URL to use in resolving the URL of the WSDL document.

See Also: "Options to Facilitate Web Services Call-Outs" on page 6-41

Server-Side Java Invocation (Call-in)

The server-side Java call-in functionality allows JPublisher to publish Java classes in the database for client-side invocation. JPublisher generates Java clients to invoke server-side Java.

In Oracle Database 10*g* release 1 (10.1), the JPublisher option for server-side call-in is `-java`. JPublisher generates a Java client that uses the dynamic invocation interface, `oracle.jpub.runtime.Client`, that is provided in the JPublisher run time, to invoke the `oracle.jpub.runtime.Server` server-side class, which in turn calls the desired Java stored procedure. The `Client` and `Server` interfaces are a part of the JPublisher run time. Only static methods with serializable parameters and return types are supported. Beginning with Oracle Database 10*g* release 1 (10.1), `oracle.jpub.runtime.Server` is located in the database.

In Oracle Database 10*g* release 2 (10.2), for server-side call-ins, JPublisher generates a PL/SQL wrapper for the stored procedure and the Java client that calls this PL/SQL wrapper. It supports both static and instance methods. The parameter and return types supported are primitive types, Java Beans, Serializable objects, and Oracle Java Database Connectivity (JDBC) types, typically those with the package name `oracle.sql`.

In Oracle Database 10*g* release 2 (10.2), the `-java` option is deprecated and the JPublisher option for server-side call-in is `-dbjava`. However, the `-java` option is still supported for backward compatibility. When the `-compatible` option is set to `10.1`, `-dbjava` behaves same as `-java`.

See Also: "Generated Code: Server-Side Java Call-in" on page A-11

Features to Filter JPublisher Output

JPublisher provides some options that allow you to filter what JPublisher produces. For example, publishing just a subset of stored procedures from a package, filtering generated code according to parameter modes or parameter types, and ensuring that generated classes follow the JavaBeans specification.

The following sections provide details:

- Publishing a Specified Subset of Functions or Procedures
- Publishing Functions or Procedures According to Parameter Modes or Types
- Ensuring that Generated Methods Adhere to the JavaBeans Specification

Publishing a Specified Subset of Functions or Procedures

Extended functionality of the `-sql` option enables you to publish just a subset of the stored functions or procedures from a package or from the SQL top level.

Recall that the following syntax results in publication of all the stored procedures of a package:

```
-sql=plsql_package
```

To publish only a subset of the stored procedures of the package, use the following syntax:

```
-sql=plsql_package(proc1+proc2+proc3+...)
```

You can also specify the subset according to stored procedure names and argument types. Instead of just specifying `proc1`, you can specify the following:

```
proc1(sqltype1, sqltype2, ...)
```

> **See Also:** "Declaration of Object Types and Packages to Translate" on page 6-14

Publishing Functions or Procedures According to Parameter Modes or Types

In some cases, particularly for code generation for Web services, not all parameter modes or types are supported in method signatures or attributes for the target usage of your code. The `-filtermodes` and `-filtertypes` options are introduced to allow you to filter generated code as needed, according to parameter modes, parameter types, or both.

For each option setting, start with a 1 to include all possibilities by default, that is no filtering is done. Then list specific modes or types that you want to exclude each followed by a minus sign (–). For example:

```
-filtertypes=1,.ORADATA-,.ORACLESQL-

-filtermodes=1,out-,inout-
```

Alternatively, you can start with a 0 to filter everything out. Then list specific modes or types that you want to allow each followed by a plus sign (+). For example:

```
-filtertypes=0,.CURSOR+,.INDEXBY+

-filtermodes=0,in+,return+
```

See Also: "Method Filtering According to Parameter Modes" on page 6-28 and "Method Filtering According to Parameter Types" on page 6-29

Ensuring that Generated Methods Adhere to the JavaBeans Specification

The `-generatebean` option is a flag that you can use to ensure that generated classes follow the JavaBeans specification. The default setting is `-generatebean=false`.

With the `-generatebean=true` setting, some generated methods are renamed so that they are not assumed to be JavaBean property getter or setter methods. This is accomplished by prefixing the method names with an underscore (_).

See Also: "Code Generation Adherence to the JavaBeans Specification" on page 6-30

Backward Compatibility and Migration

This section discusses issues of backward compatibility, compatibility between Java Development Kit (JDK) versions, and migration between Oracle8*i*, Oracle9*i*, and Oracle Database 10*g* releases of the JPublisher utility.

Default option settings and some features of the generated code changed in Oracle9*i*. If you have created an application using an Oracle8*i* implementation of JPublisher, you probably will not be able to rerun JPublisher in Oracle Database 10*g* (or Oracle9*i*) and have the generated classes still work within your application.

In addition, there were changes in JPublisher functionality between Oracle9*i* and Oracle Database 10*g*, although to a lesser degree. The main difference is that `.sqlj` files are no longer visibly generated by default, but you can change this behavior through a JPublisher setting.

The following subsections cover the details:

- JPublisher Backward Compatibility
- Changes in JPublisher Behavior Between Oracle Database 10g Release 1 and Release 2
- Changes in JPublisher Behavior Between Oracle9i Database and Oracle Database 10g
- Changes in JPublisher Behavior Between Oracle8i Database and Oracle9i Database
- JPublisher Backward-Compatibility Modes and Settings

JPublisher Backward Compatibility

The JPublisher run time is packaged with JDBC in the `classes12.jar` or `ojdbc14.jar` library. Code generated by an earlier version of JPublisher is compatible as follows:

- It can continue to run with the current release of the JPublisher run time.
- It can continue to compile against the current release of the JPublisher run time.

If you use an earlier release of the JPublisher run time and the Oracle JDBC drivers in generating code, then you can compile the code against that version of the JPublisher run time. Specifically, for use with an Oracle8*i* JDBC driver, JPublisher can generate code that implements the deprecated `CustomDatum` interface instead of the `ORAData` interface that replaced it.

Changes in JPublisher Behavior Between Oracle Database 10*g* Release 1 and Release 2

In Oracle Database 10*g* release 2 (10.2), JPublisher adds the following new APIs for Java classes generated for PL/SQL:

- `<init>(javax.sql.DataSource)`

  A constructor that takes a `java.sql.DataSource` object as argument

- `setDataSource(javax.sql.DataSource)`

  A method to set the data source that takes a `java.sql.DataSource` object as argument

These methods allow the Java wrapper to acquire a JDBC connection from the data source provided as argument.

JPublisher supports the use of SQL URI types that store URLs, referred to as **data links**. In Oracle Database 10*g* release 1 (10.1), JPublisher maps the SQL URI type, `SYS.URITYPE`, and the subtypes, `SYS.DBURITYPE`, `SYS.XDBURITYPE`, and `SYS.HTTPURITYPE`, to `java.net.URL`. When SQL URI types are used as PL/SQL stored procedures or SQL statement parameter and return types, this mapping works. However, when a SQL URI type is used as an attribute of a SQL type or element of a SQL array type, the mapping raises `ClassCastException` at run time.

To overcome this issue, in Oracle Database 10*g* release 2 (10.2), the SQL URI types are mapped to the `ORAData` subclasses that are generated by JPublisher. This is similar to the mapping used for user-defined SQL object types. You can also force JPublisher to map a SQL URI type to `java.net.URL` by specifying the following:

```
-adddefaulttypemap=
SYS.URITYPE:java.net.URL:VARCHAR2:SYS.URIFACTORY.GETURI:SYS.SQLJUTL.URI2VCHAR
-adddefaulttypemap=
SYS.DBURITYPE:java.net.URL:VARCHAR2:SYS.DBURITYPE.CREATEURI:SYS.SQLJUTL.URI2VCHAR
-adddefaulttypemap=
SYS.XDBURITYPE:java.net.URL:VARCHAR2:SYS.XDBURITYPE.CREATEURI:SYS.SQLJUTL.URI2VCHA
R
-adddefaulttypemap=
SYS.HTTPURITYPE:java.net.URL:VARCHAR2:SYS.HTTPURITYPE.CREATEURI:SYS.SQLJUTL.URI2VC
HAR
```

This includes the specification of data conversion functions.

> **See Also:** "Type Mapping Support Through PL/SQL Conversion Functions" on page 3-16 and "Type Map Options" on page 6-25

Changes in JPublisher Behavior Between Oracle9*i* Database and Oracle Database 10*g*

Regarding backward compatibility, a key difference in JPublisher behavior between Oracle9*i* Database and Oracle Database 10*g* is that now, by default, SQLJ source code is translated automatically and the `.sqlj` source files are invisible to the user.

> **See Also:** "JPublisher Usage of the SQLJ Implementation" on page 1-4

In addition, note the following changes in JPublisher behavior in Oracle Database 10*g*:

- In Oracle9*i* Database, JPublisher generates SQLJ classes with a `protected` constructor with a boolean argument to specify whether the object must be initialized. For example:

  ```
  protected BaseClass(boolean init) { ... }
  ```

This constructor is removed in Oracle Database 10*g*, because it conflicts with the constructor generation for a SQL object type with `BOOLEAN` attributes.

- In Oracle Database 10*g*, `SMALLINT` is mapped to `int` instead of `short` in Java.

Changes in JPublisher Behavior Between Oracle8*i* Database and Oracle9*i* Database

Note the following changes in JPublisher behavior, beginning with Oracle9*i* Database:

- By default, JPublisher does not declare the inner SQLJ connection context class `_Ctx` for every object type. Instead, it uses the `sqlj.runtime.ref.DefaultContext` connection context class throughout.

  In addition, user-written code must call the `getConnectionContext()` method to have a connection context instance, instead of using the `_ctx` connection context field declared in code generated by Oracle8*i* versions of JPublisher.

  > **See Also:** "Connection Contexts and Instances in SQLJ Classes" on page 4-10

- Even with the `-methods=true` setting, non-SQLJ classes are generated if the underlying SQL object type or PL/SQL package does not define any methods. However, a setting of `-methods=always` always results in SQLJ classes being produced.
- By default, JPublisher generates code that implements the `oracle.sql.ORAData` interface instead of the deprecated `oracle.sql.CustomDatum` interface.
- By default, JPublisher places generated code into the current directory, rather than into a package/directory hierarchy under the current directory.

Changes in User-Written Subclasses of JPublisher-Generated Classes

If you provided user-written subclasses for classes generated by an Oracle8*i* version of JPublisher, then you need to be aware that several relevant changes were introduced in Oracle9*i* Database related to JPublisher code generation. You must make changes in any applications that have Oracle8*i* functionality if you want to use them in Oracle9*i* Database or Oracle Database 10*g*.

Note: If you use the `-compatible=8i` or `-compatible=both8i` option setting, then you will not see the changes discussed here and your application will continue to build and work as before. For more information, refer to "Backward Compatibility Option" on page 6-46.

However, it is advised that you make the transition to the Oracle Database 10*g* JPublisher functionality, which insulates your user code from implementation details of JPublisher-generated classes.

You need to make the following changes to use your code in Oracle9*i* Database or Oracle Database 10*g*:

- Replace any use of the declared `_ctx` connection context field with use of the provided `getConnectionContext()` method. The `_ctx` field is no longer supported.

- Replace the explicit implementation of the `create()` method with a call to a superclass `create()` method, and use `ORAData` instead of `CustomDatum` as the return type.

  In the example that follows, assume that `UserClass` extends `BaseClass`. Instead of writing the following method in `UserClass`:

  ```
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    UserClass o = new UserClass();
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
    o._ctx = new _Ctx(((STRUCT) d).getConnection());
    return o;
  }
  ```

  Supply the following:

  ```
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    return create(new UserClass(),d,sqlType);
  }
  ```

  Alternatively, if the class is part of an inheritance hierarchy, then write the following:

  ```
  protected ORAData createExact(Datum d, int sqlType) throws SQLException
  {
    return create(new UserClass(),d,sqlType);
  }
  ```

- In addition to the `getConnectionContext()` method, JPublisher provides a `getConnection()` method that can be used to obtain the JDBC connection instance associated with the object.

JPublisher Backward-Compatibility Modes and Settings

JPublisher supports settings for backward-compatibility modes through the `-compatible` option. At the most elementary level, this includes a setting to explicitly generate `.sqlj` files, which are transparent to users in Oracle Database 10*g* by default. There are also Oracle9*i* and Oracle8*i* compatibility modes, involving differences in the generated code itself as well as the creation of visible `.sqlj` files. The following topics are discussed:

- Explicit Generation of .sqlj Files
- Oracle9i Compatibility Mode
- Oracle8i Compatibility Mode
- Individual Settings to Force Oracle8i JPublisher Behavior

See Also: "Backward Compatibility Option" on page 6-46

Explicit Generation of .sqlj Files

In Oracle Database 10*g*, if you want to avoid automatic SQLJ translation so that JPublisher generates `.sqlj` files that you can work with directly, then you can use the `-compatible=sqlj` JPublisher setting.

Note: In Oracle Database 10*g*, you do not have to invoke the SQLJ translator directly to explicitly translate `.sqlj` files. You can use the JPublisher `-sqlj` option instead.

See Also: "Option to Access SQLJ Functionality" on page 6-45

Oracle9*i* Compatibility Mode

The `-compatible=9i` JPublisher option setting enables the Oracle9*i* compatibility mode. In this mode, JPublisher generates code that is compatible with Oracle9*i* SQLJ and JDBC releases. In addition, JPublisher typically produces `.sqlj` files that are visible to the user, as is the case with Oracle9*i* JPublisher.

JPublisher has the following functionality in the Oracle9*i* compatibility mode:

- In SQLJ classes, JPublisher generates a `protected` constructor with a `boolean` argument that specifies whether the object must be initialized, as it does in Oracle9*i*:

  ```
  protected BaseClass(boolean init) { ... }
  ```

  This constructor is removed in Oracle Database 10*g*

  See Also: "Changes in JPublisher Behavior Between Oracle9i Database and Oracle Database 10g" on page 5-6

- The mapping in Java from `SMALLINT` reverts from `int`, which is the mapping in Oracle Database 10*g*, to `short`.

Oracle8*i* Compatibility Mode

Either the `-compatible=both8i` or `-compatible=8i` JPublisher setting enables the Oracle8*i* compatibility mode. In this mode, JPublisher generates code that is compatible with Oracle8*i* SQLJ and JDBC releases. In addition, JPublisher typically produces `.sqlj` files visible to the user, as is the case with Oracle8*i* JPublisher.

However, for the use of this mode to be permissible, at least one of the following circumstances must hold:

- You translate JPublisher-generated `.sqlj` files with the default SQLJ `-codegen=oracle` setting.
- The JPublisher-generated code runs under JDK 1.2 or later and uses the SQLJ `runtime12.jar` library or runs in the Oracle Database 10*g* release of the server-side Oracle Java virtual machine (JVM).
- You run JPublisher with the `-methods=false` or `-methods=none` setting.

Note the following functionality in the Oracle8*i* compatibility mode:

- JPublisher generates code that implements the deprecated `CustomDatum` and `CustomDatumFactory` interfaces instead of the `ORAData` and `ORADataFactory` interfaces, as with the `-compatible=customdatum` setting. In addition, if you choose the `-compatible=both8i` setting, then the generated code also implements the `ORAData` interface, though not `ORADataFactory`.
- With the `-methods=true` setting, JPublisher always generates a SQLJ class for a SQL object type, even if the object type does not define any methods. This is the same as using the `-methods=always` setting.

- JPublisher generates connection context declarations and connection context instances on every object wrapper class, as follows:

  ```
  #sql static context _Ctx;
  protected _Ctx _ctx;
  ```

  This is the same as the `-context=generated` setting.

- JPublisher provides a constructor in the wrapper class that takes a generic `ConnectionContext` instance, which is an instance of any class implementing the standard `sqlj.runtime.ConnectionContext` interface, as input. In Oracle Database 10*g*, the constructor accepts only a `DefaultContext` instance or an instance of the class specified through the `-context` option when JPublisher was run.

- JPublisher does not provide an API for releasing a connection context instance that has been created implicitly on a JPublisher object.

  By contrast, the JPublisher utility in Oracle Database 10*g* provides both a `setConnectionContext()` method for explicitly setting the connection context instance for an object, and a `release()` method for releasing an implicitly created connection context instance of an object.

If you must choose the Oracle8*i* compatibility mode, then it is advisable to use the `-compatible=both8i` setting. This permits your application to work in a middle-tier environment, such as Oracle Application Server, in which JDBC connections are obtained through data sources and will likely be wrapped using `oracle.jdbc.Oracle`*Xxxx* interfaces. `CustomDatum` implementations do not support such wrapped connections.

> **Note:** The `-compatible=both8i` setting requires a JDBC implementation from Oracle9*i* release 1 (9.0.1) or later.

Oracle8*i* compatibility mode is now the only way for a `_ctx` connection context instance to be declared in JPublisher-generated code. No other option setting accomplishes this particular Oracle8*i* behavior. The `_ctx` instance may be useful if you have legacy code that depends on it, but otherwise you should obtain connection context instances through the `getConnectionContext()` method.

Individual Settings to Force Oracle8*i* JPublisher Behavior

The individual option settings detailed in Table 5–1 will produce results, most of which are similar to those produced when using JPublisher in the Oracle8*i* compatibility mode.

Table 5–1 JPublisher Backward Compatibility Options

Option Setting	Behavior
`-context=generated`	This setting results in the declaration of an inner class, `_Ctx`, for SQLJ connection contexts. This is used instead of the default `DefaultContext` class or user-specified connection context classes.
`-methods=always`	This setting forces generation of SQLJ classes, in contrast to non-SQLJ classes, for all JPublisher-generated classes, whether or not the underlying SQL objects or packages define any methods.

Table 5–1 (Cont.) JPublisher Backward Compatibility Options

Option Setting	Behavior
`-compatible=customdatum`	For Oracle-specific wrapper classes, this setting results in JPublisher implementing the deprecated `oracle.sql.CustomDatum` and `CustomDatumFactory` interfaces instead of the `oracle.sql.ORAData` and `ORADataFactory` interfaces.
`-dir=.`	Setting this option to a period (`.`), results in the generation of output files into a hierarchy under the current directory, as was the default behavior in Oracle8*i*.

For detailed descriptions of these options, refer to the following:

- "SQLJ Connection Context Classes" on page 6-20
- "Generation of Package Classes and Wrapper Methods" on page 6-32
- "Backward-Compatible Oracle Mapping for User-Defined Types" on page 6-46
- "Output Directories for Generated Source and Class Files" on page 6-40

See Also: "Oracle8i Compatibility Mode" on page 5-9

6

Command-Line Options and Input Files

This chapter describes the usage and syntax details of JPublisher option settings and input files to specify program behavior. It is organized into the following sections:

- JPublisher Options
- JPublisher Input Files

JPublisher Options

The following sections list and discuss JPublisher command-line options:

- JPublisher Option Summary
- JPublisher Option Tips
- Notational Conventions
- Options for Input Files and Items to Publish
- Connection Options
- Options for Data Type Mappings
- Type Map Options
- Java Code-Generation Options
- PL/SQL Code Generation Options
- Input/Output Options
- Options to Facilitate Web Services Call-Outs
- Option to Access SQLJ Functionality
- Backward Compatibility Option
- Java Environment Options
- SQLJ Migration Options

JPublisher Option Summary

Table 6–1 summarizes JPublisher options. For default values, the abbreviation, NA, means not applicable. The Category column refers to the corresponding conceptual area, indicating the section of this chapter where the option is discussed.

Table 6–1 Summary of JPublisher Options

Option Name	Description	Default Value	Category
`-access`	Determines the access modifiers that JPublisher includes in generated method definitions.	`public`	Java code generation
`-adddefaulttypemap`	Appends an entry to the JPublisher default type map.	NA	Type maps
`-addtypemap`	Appends an entry to the JPublisher user type map.	NA	Type maps
`-builtintypes`	Specifies the data type mappings, `jdbc` or `oracle`, for built-in data types that are not numeric or large object (LOB).	`jdbc`	Data type mappings
`-case`	Specifies the case of Java identifiers that JPublisher generates.	`mixed`	Java code generation
`-classpath`	Adds to the Java classpath for JPublisher to resolve Java source and classes during translation and compilation.	Empty	Java environment
`-compatible`	Specifies a compatibility mode and modifies the behavior of `-usertypes=oracle`. **See Also:** "JPublisher Backward-Compatibility Modes and Settings" on page 5-8	`oradata`	Backward compatibility
`-compile`	Determines whether to proceed with Java compilation or suppress it. This option also affects SQLJ translation for backward-compatibility modes.	`true`	Input/output
`-compiler-executable`	Specifies a Java compiler version, in case you want a version other than the default.	NA	Java environment
`-context`	Specifies the class that JPublisher uses for SQLJ connection contexts. This can be the `DefaultContext` class, a user-specified class, or a JPublisher-generated inner class.	`DefaultContext`	Connection
`-defaulttypemap`	Sets the default type map that JPublisher uses.	Refer to "JPublisher User Type Map and Default Type Map" on page 3-5.	Type maps
`-d`	Specifies the root directory for placement of compiled class files.	Empty (all files directly present in the current directory)	Input/output
`-dir`	Specifies the root directory for placement of generated source files.	Empty (all files directly present in the current directory)	Input/output
`-driver`	Specifies the driver class that JPublisher uses for Java Database Connectivity (JDBC) connections to the database.	`oracle.jdbc.OracleDriver`	Connection
`-encoding`	Specifies the Java encoding of JPublisher input and output files.	The value of the system property `file.encoding`	Input/output

Table 6–1 (Cont.) Summary of JPublisher Options

Option Name	Description	Default Value	Category
`-endpoint`	Specifies a Web service endpoint. This option is used in conjunction with the `-proxywsdl` option.	NA	Web services
`-filtermodes`	Filters code generation according to specified parameter modes.	NA	Java code generation
`-filtertypes`	Filters code generation according to specified parameter types.	NA	Java code generation
`-generatebean`	Ensures that generated code conforms to the JavaBeans specification.	`false`	Java code generation
`-genpattern`	Defines naming patterns for generated code.	NA	Java code generation
`-gensubclass`	Specifies whether and how to generate stub code for user subclasses.	`true`	Java code generation
`-httpproxy`	Specifies a proxy URL to resolve the URL of a Web Services Description Language (WSDL) document for access through a firewall. This option is used in conjunction with the `-proxywsdl` option.	NA	Web services
`-input` or `-i`	Specifies a file that lists the types and packages that JPublisher translates.	NA	Input files/items
`-java`	Specifies server-side Java classes for which JPublisher generates client-side classes.	NA	Input files/items
`-lobtypes`	Specifies the `jdbc` or `oracle` data type mapping that JPublisher uses for `BLOB` and `CLOB` types.	`oracle`	Data type mappings
`-mapping`	Specifies the mapping that generated methods support for object attribute types and method argument types. **Note**: This option is deprecated in favor of the "*XXX*`types`" mapping options, but is supported for backward compatibility.	`objectjdbc`	Data type mappings
`-methods`	Determines whether JPublisher generates wrapper methods for stored procedures of translated SQL objects and PL/SQL packages. This option also determines whether JPublisher generates SQLJ classes or non-SQLJ classes, and whether it generates PL/SQL wrapper classes at all. There are settings to specify whether overloaded methods are allowed.	`all`	Java code generation
`-numbertypes`	Specifies the data type mappings, such as `jdbc`, `objectjdbc`, `bigdecimal`, or `oracle`, that JPublisher uses for numeric data types.	`objectjdbc`	Data type mappings
`-omit_schema_names`	Instructs JPublisher not to include the schema in SQL type name references in generated code.	Disabled (schema included in type names)	Java code generation

Table 6–1 (Cont.) Summary of JPublisher Options

Option Name	Description	Default Value	Category
`-outarguments`	Specifies the holder type, such as arrays, Java API for XML-based Remote Procedure Call (JAX-RPC) holders, or function returns, for Java implementation of PL/SQL output parameters.	`array`	Java code generation
`-overwritedbtypes`	Specifies whether to ignore naming conflicts when creating SQL types.	`true`	PL/SQL code generation
`-package`	Specifies the name of the Java package into which JPublisher generates Java wrapper classes.	NA	Java code generation
`-plsqlfile`	Specifies a wrapper script to create and a dropper script to drop SQL conversion types for PL/SQL types and the PL/SQL package that JPublisher will use for generated PL/SQL code.	`plsql_wrapper.sql`, `plsql_dropper.sql`	PL/SQL code generation
`-plsqlmap`	Specifies whether to generate PL/SQL wrapper functions for stored procedures that use PL/SQL types.	`true`	PL/SQL code generation
`-plsqlpackage`	Specifies the PL/SQL package into which JPublisher generates PL/SQL code, such as call specifications, conversion functions, and wrapper functions.	`JPUB_PLSQL_WRAPPER`	PL/SQL code generation
`-props` or `-p`	Specifies a file that contains JPublisher options in addition to those listed on the command line.	NA	Input files/items
`-proxyclasses`	Specifies Java classes for which JPublisher generates wrapper classes and PL/SQL wrappers according to the `-proxyopts` setting. For Web services, you will typically use `-proxywsdl` instead, which uses `-proxyclasses` behind the scenes.	NA	Web services
`-proxyopts`	Specifies required layers of Java and PL/SQL wrappers and additional related settings. Is used as input for the `-proxywsdl` and `-proxyclasses` options.	`jaxrpc`	Web services
`-proxywsdl`	Specifies the URL of a WSDL document for which Web services client proxy classes and associated Java wrapper classes are generated along with PL/SQL wrappers.	NA	Web services
`-serializable`	Specifies whether the code generated for object types implements the `java.io.Serializable` interface.	`false`	Java code generation
`-sql` or `-s`	Specifies object types and packages, or subsets of packages, for which JPublisher generates Java classes, and optionally subclasses and interfaces.	NA	Input files/items

Table 6–1 (Cont.) Summary of JPublisher Options

Option Name	Description	Default Value	Category
`-sqlj`	Specifies SQLJ option settings for the JPublisher invocation of the SQLJ translator.	NA	SQLJ
`-sqlstatement`	Specifies SQL queries or data manipulation language (DML) statements for which JPublisher generates Java classes, and optionally subclasses and interfaces, with appropriate methods.	NA	Input files/items
`-style`	Specifies the name of a "style file" for Java-to-Java type mappings.	NA	Data type mappings
`-sysuser`	Specifies the name and password for a superuser account that can be used to grant permissions to execute wrappers that access Web services client proxy classes in the database.	NA	Web services
`-tostring`	Specifies whether to generate a `toString()` method for object types.	`false`	Java code generation
`-typemap`	Specifies the JPublisher type map.	Empty	Type maps
`-types`	Specifies object types for which JPublisher generates code. **Note**: This option is deprecated in favor of `-sql`, but is supported for backward compatibility.	NA	Input files/items
`-url`	Specifies the URL that JPublisher uses to connect to the database.	`jdbc:oracle:oci:@`	Connection
`-user` or `-u`	Specifies an Oracle user name and password for connection.	NA	Connection
`-usertypes`	Specifies the `jdbc` or `oracle` type mapping that JPublisher uses for user-defined SQL types.	`oracle`	Data type mappings
`-vm`	Specifies a Java version, in case you want a version other than the default.	NA	Java environment

JPublisher Option Tips

Be aware of the following usage notes for JPublisher options:

- JPublisher always requires the `-user` option or its shorthand equivalent `-u`.
- Options are processed in the order in which they appear. Options from an `INPUT` file are processed at the point where the `-input` or `-i` option occurs. Similarly, options from a properties file are processed at the point where the `-props` or `-p` option occurs.
- As a rule, if a particular option appears more than once, JPublisher uses the value from the last occurrence. However, this is *not* true for the following options, which are cumulative:

  `-sql`

  `-types`, which is deprecated

  `-java`

`-addtypemap` or `-adddefaulttypemap`

`-style`

- In general, separate options and corresponding option values by an equal sign (=). However, when the following options appear on the command line, you can also use a space as a separator:

  `-sql` or `-s`, `-user` or `-u`, `-props` or `-p`, and `-input` or `-i`

- With the `-sqlj` option, you *must* use a space instead of an equal sign, because SQLJ settings following the `-sqlj` option use equal signs. Consider the following example, where each entry after "`-sqlj`" is a SQLJ option:

```
% jpub -user=scott/tiger -sql=PERSON:Person -sqlj -optcols=true -optparams=true
       -optparamdefaults=datatype1(size1),datatype2(size)
```

- It is advisable to specify a Java package for generated classes with the `-package` option, either on the command line or in a properties file. For example, you could enter the following on the command line:

```
% jpub -sql=Person -package=e.f ...
```

  Alternatively, you could enter the following in the properties file:

```
jpub.sql=Person
jpub.package=e.f
...
```

  These statements direct JPublisher to create the class `Person` in the Java package `e.f`, that is, to create the class `e.f.Person`.

  See Also: "Properties File Structure and Syntax" on page 6-51

- If you do not specify a type or package in the `INPUT` file or on the command line, then JPublisher translates all types and packages in the user schema according to the options specified on the command line or in the properties file.

Notational Conventions

The JPublisher option syntax used in the following sections uses the following notational conventions:

- Braces `{...}` enclose a list of possible values. Specify only one of the values within the braces.
- A vertical bar | separates alternatives within braces.
- Terms in *`italics`* are for user input. Specify an actual value or string.
- Terms in **boldface** indicate default values.
- Square brackets `[...]` enclose optional items. In some cases, however, square brackets or parentheses are part of the syntax and must be entered verbatim. In this case, this manual uses boldface: **`[...]`** or **`(...)`**.
- Ellipsis points `...` immediately following an item, or items enclosed in brackets, mean that you can repeat the item any number of times.
- Punctuation symbols other than those described in this section are entered as shown in this manual. These include "`.`" and "`@`", for example.

Options for Input Files and Items to Publish

This section documents the following JPublisher options that specify key input, either JPublisher input files, such as `INPUT` files or properties files, or items to publish, such as SQL objects, PL/SQL packages, SQL queries, SQL DML statements, or server-side Java classes:

- Options for input files: `-input`, `-props`
- Options for items to publish: `-java`, `-sql`, `-sqlstatement`, `-types`

These options are discussed in alphabetic order.

File Containing Names of Objects and Packages to Translate

The `-input` option specifies the name of a file from which JPublisher reads the names of SQL or PL/SQL entities or server-side Java classes to publish, along with any related information or instructions. JPublisher publishes each item in the list. You can think of the `INPUT` file as a makefile for type declarations, which lists the types that need Java class definitions.

The syntax of the `-input` option is as follows:

```
-input=filename
-i filename
```

Both formats are synonymous. The second one is a convenient command-line shortcut.

In some cases, JPublisher may find it necessary to translate some additional classes that do not appear in the `INPUT` file. This is because JPublisher analyzes the types in the `INPUT` file for dependencies before performing the translation and translates other types as necessary.

If you do not specify any items to publish in an `INPUT` file or on the command line, then JPublisher translates all user-defined SQL types and PL/SQL packages declared in the database schema to which it is connected.

> **See Also:** "Translating Additional Types" on page 6-55 and "INPUT File Structure and Syntax" on page 6-52

Declaration of Server-Side Java Classes to Publish

The `-java` option enables you to create client-side stub classes to access server-side classes. This is an improvement over earlier JPublisher releases in which calling Java stored procedures and functions from a database client required JDBC calls to associated PL/SQL wrappers.

The syntax of the `-java` option is as follows:

```
-java=class_or_package_list
```

The functionality of the `-java` option mirrors that of the `-sql` option. It creates a client-side Java stub class to access a server-side Java class, in contrast to creating a client-side Java class to access a server-side SQL object or PL/SQL package.

When using the `-java` option, specify a comma-delimited list of server-side Java classes or packages.

Notes:

- To use the `-java` option, you must also specify the `-user` and `-url` settings for a database connection.
- Functionality of the `-java` option requires the `sqljutl.jar` library to be loaded in the database. For more information, refer to "Required Database Setup" on page 1-8.
- It is advisable to use the same Java Development Kit (JDK) on the client as on the server.

For example:

```
-java=foo.bar.Baz,foo.baz.*
```

Or, to specify the client-side class name corresponding to `Baz`, instead of using the server-side name by default:

```
-java=foo.bar.Baz:MyBaz,foo.baz.*
```

This setting creates `MyBaz` and not `foo.bar.MyBaz`.

or:

```
-java=foo.bar.Baz:foo.bar.MyBaz,foo.baz.*
```

You can also specify a schema:

```
-java=foo.bar.Baz@SCOTT
```

If you specify the schema, then only that schema is searched. If you do not specify a schema, then the schema of the logged-in user, according to the `-user` option setting, is searched. This is the most likely scenario.

As an example, assume that you want to call the following method on the server:

```
public String oracle.sqlj.checker.JdbcVersion.to_string();
```

Use the following `-java` setting:

```
-java=oracle.sqlj.checker.JdbcVersion
```

Note: If JPublisher cannot find a specified class in the schema, a specified schema or the schema of the logged-in user, then it uses the `Class.forName()` method to search for the class among system classes in the Java virtual machine (JVM), typically Java run-time environment (JRE) or JDK classes.

Code Generation for -java Option When you use the `-java` option, generated code uses the following API:

```
public class Client
{
   public static String getSignature(Class[]);
   public static Object invoke(Connection, String, String,
                               String, Object[]);
   public static Object invoke(Connection, String, String,
                               Class[], Object[]);
}
```

Classes for the API are located in the `oracle.jpub.reflect` package, so client applications must import this package.

For a setting of `-java=oracle.sqlj.checker.JdbcVersion`, JPublisher-generated code includes the following call:

```
Connection conn = ...;
String serverSqljVersion = (String)
          Client.invoke(conn, "oracle.sqlj.checker.JdbcVersion",
          "to_string", new Class[]{}, new Object[]{});
```

The `Class[]` array is for the method parameter types, and the `Object[]` array is for the parameter values. In this case, because `to_string` has no parameters, the arrays are empty.

Note the following:

- Any serializable type, such as `int[]` or `String[]`, can be passed as an argument.
- The semantics of this API are different from the semantics for invoking Java stored procedures or functions through a PL/SQL wrapper, in the following ways:
  - Arguments cannot be `OUT` or `IN OUT`. Returned values must be part of the function result.
  - Exceptions are properly returned.
  - Method invocation uses invoker's rights. There is no tuning to obtain definer's rights.

    See Also: *Oracle Database Java Developer's Guide* for information about invoker's rights and definer's rights

Declaration of Server-Side Java Classes to Publish

The related options for publishing a server-side Java class are:

```
-dbajva=class_list

-proxyopts=single|multiple|static|arrayin|arrayout|arrayinout|arrayall|noload

-compatible=10.1

-sysuser=user/password

-plsqlfile=wrapper[,dropper]

-plsqlpackage=name
```

Oracle Database 10*g* release 1 (10.1) introduces the `-java` option to publish server-side Java classes. Oracle Database 10*g* release 2 (10.2) introduces a new approach toward server-side Java class publishing. The `-dbjava` option publishes a server-side Java class into PL/SQL, or into client-side Java class. The `class_list` specification is a comma-delimited list of server-side classes at the specified server. The `class_list` item is of the form `classname[:name[#interface]]`. It can also be a package name. Consider the option:

```
-dbajva=classname[:name[#interface]]
```

If `name` is not specified, then the server-side Java class, `classname`, is published into PL/SQL, else into the client-side Java class, `name`. If `interface` is specified, then the interface file is generated for the client-side Java class.

When used with `-dbjava`, the `-proxyopts` option indicates whether to map instance methods using a singleton instance or using multiple instances, and also whether to map methods with array parameters assuming arrays as `IN`, `OUT`, `IN OUT`, or all the modes. The `-proxyopts=static` setting specifies that only static methods should be published. The default setting, `-proxyopts=single,arrayin`, indicates that instance methods are called using a singleton instance and array parameters are considered as input. The `-proxyopts=noload` setting forbids JPublisher from loading the generated PL/SQL and Java stored procedure wrappers.

The `-compatible=10.1` option makes `-dbjava` equivalent to `-java`.

Declaration of Server-Side Java Classes to Publish

The related options for publishing server-side Java class are:

```
-proxyclasses=class_or_jar_list

-proxyopts=single|multiple|static|arrayin|arrayout|arrayinout|arrayall

-plsqlfile=wrapper[,dropper]

-plsqlpackage=name
```

The `-proxyclasses` option is similar to `-dbjava`. While `-dbjava` requires that the classes to be published exist in the database, `-proxyclasses` requires that the classes appear in the classpath. Typically, by using `-proxyclasses`, you can load the exposed classes and the generated wrappers into the database later.

The `-proxyclasses` option generates only a PL/SQL wrapper. Unlike `-dbjava`, it will not generate client-side Java code for a server-side Java class. Also, unlike `-dbjava`, `-proxyclasses` does not load the generated Java stored procedure into the database.

You can use the `-proxyclasses` option to specify a comma-delimited list of Java classes, either loose classes or Java Archive (JAR) files, for which JPublisher creates PL/SQL wrappers. Depending on the situation, JPublisher can also create Java wrapper classes to afford access from PL/SQL. Each of the classes processed must have either public, static methods or, for classes in which you want to publish instance methods, a `public` zero-argument constructor.

To summarize, the following are generated for each class being processed, depending on the `-proxyopts` option settings:

- A PL/SQL wrapper to allow access from PL/SQL. This is always generated.
- A wrapper class to expose Java instance methods as static methods, if there are any instance methods to publish.

Instance methods must be exposed as static methods to allow access from PL/SQL. A wrapper class is also necessary if the wrapped class uses anything other than Java primitive types in the method calling sequences.

While using the `-proxyclasses` option directly, you can specify JAR files and Java classes that exist in the classpath. Classes and JAR files can be specified as follows:

- Class name, such as `foo.bar.Baz` or `foo.bar.Baz.class`
- Package name, such as `foo.bar.*`, for `@server` mode only
- JAR or ZIP file name, such as `foo/bar/baz.jar` or `Baz.zip`
- JAR or ZIP file name followed by parenthesized list of classes or packages, such as `baz.jar (foo.MyClass1, foo.bar.MyClass2, foo1.*)`

Settings for Java and PL/SQL Wrapper Generation

The `-proxyopts` option is used as input by the `-dbjava`, `-proxywsdl`, and `-proxyclasses` options and specifies JPublisher behavior in generating wrapper classes and PL/SQL wrappers for server-side Java classes.

The syntax of the `-proxyopts` option is as follows:

```
-proxyopts=setting1,setting2,...
```

This option uses the basic settings, which can be used individually or in combinations. In this discussion, processed classes are the classes that are wrapped by using the `-dbjava`, `-proxywsdl`, or `-proxyclasses` options.

Where Java wrapper classes are generated, the wrapper class for a class `foo.bar.MyClass` would be `foo.bar.MyClassJPub`, unless the package is overridden by a setting of the `-package` option.

You can use the basic `-proxyopts` settings as follows:

- Use the `static` setting to specify the treatment of static methods of processed classes.

  With this setting, in the PL/SQL wrapper, a wrapper procedure is generated for each static method. Without this setting, static methods are ignored. For classes with only static methods, wrapper classes are not required for processed classes that use only Java primitive types in their method calling sequences.

- Use the `multiple` or `single` setting to specify treatment of instance methods of processed classes, where you want instance methods exposed as static methods. In either case, for each processed class, JPublisher generates an intermediate Java class that wraps instance methods with static methods, in addition to generating a PL/SQL wrapper.

  Use the `instance` setting to specify treatment of instance methods of processed classes, where you want instance methods maintained as instance methods.

  These settings function as follows:

  – `multiple`

    For each processed class, the Java wrapper class has a static equivalent for each instance method through the use of handles, which identify instances of wrapped classes.

  – `single`

    Only a single default instance of each wrapped class is used during run time. For each processed class, the Java wrapper class has static wrapper methods for instance methods without requiring the use of handles. This is the singleton mechanism.

  – `instance`

    Instance methods are wrapped as instance methods in the Java wrapper class.

    Note: The `instance` setting is not appropriate for Web services.

  The instance methods are ignored if one of these settings or a `jaxrpc` or `soap` setting, which implies `single`, is not specified. For either of these settings, only classes that provide a public zero-argument constructor are processed. You can use both settings to generate wrapper classes of both styles.

- Use the `jaxrpc` or `soap` setting to publish instance methods of Web services client proxy classes. These settings function as follows:
  - `jaxrpc`

    This is the default setting. It is a convenient setting for wrapping JAX-RPC client proxy classes, which is appropriate for use with Oracle Application Server 10*g* 10.0.1 and later releases. JPublisher creates a Java wrapper class for each processed class and also creates the PL/SQL wrapper. Client proxy classes do not have static methods to be published, and instance methods are published using the singleton mechanism by default. Therefore, when processing JAX-RPC client proxy classes, `-proxyopts=jaxrpc` implies `-proxyopts=single`. The `jaxrpc` setting also results in generation of special code that is specific to JAX-RPC clients.
  - `soap`

    This setting is equivalent to the `jaxrpc` setting, but is used for wrapping SOAP client proxy classes instead of JAX-RPC client proxy classes. This is appropriate for use with Oracle Application Server 10*g* 9.0.4 and earlier releases.

Here are some basic uses of the `-proxyopts` option:

```
-proxyopts=jaxrpc

-proxyopts=soap

-proxyopts=static

-proxyopts=static,instance

-proxyopts=single

-proxyopts=single,multiple

-proxyopts=static,multiple
```

The `static,instance` setting publishes static and instance methods. The `single,multiple` setting publishes only instance methods, using both the singleton mechanism and the handle mechanism. The `static,multiple` setting publishes static and instance methods, using the handle mechanism to expose instance methods as static methods.

> **Note:** It is typical to explicitly use the `-proxyopts` option with the `-proxyclasses` option than with the `-proxywsdl` option. For the use of `-proxywsdl` with 10.0.x releases of Oracle Application Server 10*g*, the default `-proxyopts=jaxrpc` setting is sufficient.

There are additional, advanced `-proxyopts` settings as well. The functionality of each setting is as follows:

- `noload`

  Use this to indicate that the generated code need not be loaded into the database. By default, the generated code is loaded.
- `recursive`

Use this to indicate that when processing a class that extends another class, also create PL/SQL and Java wrappers, if appropriate, for inherited methods.

- `tabfun`

  Use this with the `jaxrpc` or `soap` setting for JPublisher to generate PL/SQL table functions for the PL/SQL package for each of the wrapped Web services operations. This exposes data through database tables rather than stored procedures or functions.

- `deterministic`

  Use this to indicate in the generated PL/SQL wrapper that the wrapped methods are deterministic. This would typically be used with the `tabfun` setting. Deterministic is a PL/SQL annotation.

  See Also: *Oracle Database SQL Reference* for information about `DETERMINISTIC` functions

- `main(0,...)`

  Use this with the `static` setting to define the wrapper methods to be generated if there is a `public void String main(String[])` method in the class. A separate method is generated for each number of arguments that you want to support. You can use commas or hyphens when indicating the number of arguments, as in the following examples:

  – `main` or `main(0)` produces a wrapper method only for zero arguments.

  – `main(0,1)` produces wrapper methods for zero arguments and one argument. This is the default setting.

  – `main(0-3)` produces wrapper methods for zero, one, two, and three arguments.

  – `main(0,2-4)` produces wrapper methods for zero, two, three, and four arguments.

  The maximum number of arguments in the wrapper method for the `main()` method is according to PL/SQL limitations.

The following example uses the `jaxrpc` basic setting by default. It also uses table functions and indicates that wrapped methods are deterministic:

```
-proxyopts=tabfun,deterministic
```

The following example explicitly sets the `static` mode, processing classes that are not client proxy classes, and specifies that the generated code should not be loaded into the database:

```
-proxyopts=static,noload
```

Input Properties File

The `-props` option specifies the name of a JPublisher properties file that specifies JPublisher option settings. JPublisher processes the properties file as if its contents were inserted in sequence on the command line where the `-props` option is specified.

The syntax of the `-props` option is as follows:

```
-props=filename
-p filename
```

Both formats are synonymous. The second one is provided as a convenient command-line shortcut.

If more than one properties file appears on the command line, then JPublisher processes them with the other command-line options, in the order in which they appear.

See Also: "Properties File Structure and Syntax" on page 6-51

Note: Encoding settings, either set through the JPublisher `-encoding` option or the Java `file.encoding` setting, do not apply to Java properties files. Properties files always use the `8859_1` encoding. This is a feature of Java in general, and not of JPublisher in particular. However, you can use Unicode escape sequences in a properties file.

Declaration of Object Types and Packages to Translate

The `-sql` option is used to specify the user-defined SQL types, such as objects or collections, or the PL/SQL packages that need to be published. Optionally, you can specify the user subclasses or interfaces that should be generated. You can publish all or a specific subset of a PL/SQL package.

The syntax of the `-sql` option is as follows:

```
-sql={toplevel|object_type_and_package_translation_syntax}
-s {toplevel|object_type_and_package_translation_syntax}
```

The two formats of this option, `-sql` and `-s`, are synonymous. The `-s` format is provided as a convenient command-line shortcut.

You can use the `-sql` option when you do not need the generality of an `INPUT` file. The `-sql` option lets you list one or more database entities declared in SQL that you want JPublisher to translate. Alternatively, you can use several `-sql` options in the same command line, or several `jpub.sql` options in a properties file.

You can mix user-defined type names and package names in the same `-sql` declaration. JPublisher can detect whether each item is an object type or a package. You can also use the `-sql` option with the keyword `toplevel` to translate all top-level PL/SQL subprograms in a schema. The `toplevel` keyword is not case-sensitive.

If you do not specify any types or packages to translate in the `INPUT` file or on the command line, then JPublisher translates all the types and packages in the schema to which you are connected. In this section, the `-sql` option is explained in terms of the equivalent `INPUT` file syntax.

See Also: "Understanding the Translation Statement" on page 6-52

You can use the any of the following syntax modes:

- `-sql=`*`name_a`*

  JPublisher publishes *`name_a`*, naming the generated class according to the default settings. In an `INPUT` file, you specify this options as follows:

  ```
  SQL name_a
  ```

- `-sql=`*`name_a:class_c`*

JPublisher publishes *name_a* as the generated Java class *class_c*. In an INPUT file, you specify this options as follows:

```
SQL name_a AS class_c
```

- `-sql=`*name_a*`:`*class_b*`:`*class_c*

  In this case, *name_a* must represent an object type. JPublisher generates the Java class, *class_b*, and a stub class, *class_c*, that extends *class_b*. You provide the code for *class_c*, which is used to represent *name_a* in your Java code. In an INPUT file, you specify this options as follows:

  ```
  SQL name_a GENERATE class_b AS class_c
  ```

- `-sql=`*name_a*`:`*class_b*`#`*intfc_b*
- `-sql=`*name_a*`:`*class_b*`:`*class_c*`#`*intfc_c*

  You can use either of these syntax formats to have JPublisher generate a Java interface. This feature is particularly useful for Web services. In the first case, *class_b* represents *name_a* and implements *intfc_b*. In the second case, *class_c* represents *name_a*, extends *class_b*, and implements *intfc_c*.

  See Also: "Generation of Java Interfaces" on page 4-14

  Specify an interface for either the generated class or the user subclass, but not both. In an INPUT file, this syntax is as follows:

  ```
  SQL name_a
    [GENERATE  class_b
               [ implements intfc_b] ]
    [AS        class_c
               [ implements intfc_c ] ]
    ...
  ```

  Notes:

  - Only SQL names that are not case-sensitive are supported on the JPublisher command line. If a user-defined type was defined in a case-sensitive way in SQL, using quotes, then you must specify the name in the JPublisher INPUT file instead of specifying the user-defined type, in quotes, on the command line.
  - If your desired class and interface names follow a pattern, you can use the `-genpattern` command-line option for convenience.

If you enter more than one item for translation, then the items must be separated by commas, without any white space. This example assumes that CORPORATION is a package and that EMPLOYEE and ADDRESS are object types:

```
-sql=CORPORATION,EMPLOYEE:OracleEmployee,ADDRESS:JAddress:MyAddress
```

JPublisher interprets this command as follows:

```
SQL CORPORATION
SQL EMPLOYEE AS OracleEmployee
SQL ADDRESS GENERATE JAddress AS MyAddress
```

JPublisher performs the following actions:

- Creates a wrapper class for the `CORPORATION` package.
- Translates the `EMPLOYEE` object type as `OracleEmployee`.
- Generates an object reference class, `OracleEmployeeRef`.
- Translates `ADDRESS` as `JAddress`, but generates code and references so that `ADDRESS` objects will be represented by the `MyAddress` class.
- Generates a `MyAddress` stub, which extends `JAddress`, where you can write your custom code.
- Generates an object reference class `MyAddressRef`.

If you want JPublisher to translate all the top-level PL/SQL subprograms in the schema to which JPublisher is connected, then enter the keyword `toplevel` following the `-sql` option. JPublisher treats the top-level PL/SQL subprograms as if they were in a package. For example:

```
-sql=toplevel
```

JPublisher generates a wrapper class, `toplevel`, for the top-level subprograms. If you want the class to be generated with a different name, you can declare the name as follows:

```
-sql=toplevel:MyClass
```

Note that this is synonymous with the following `INPUT` file syntax:

```
SQL toplevel AS MyClass
```

Similarly, if you want JPublisher to translate all the top-level PL/SQL subprograms in some other schema, then enter:

```
-sql=schema_name.toplevel
```

In this example, *`schema_name`* is the name of the schema containing the top-level subprograms. In addition, there are features to publish only a subset of stored procedures in a PL/SQL package or at the top level, using the following syntax:

```
-sql=plsql_package(proc1+proc2+proc3+...)
```

Use a plus sign (+) between stored procedure names. Alternatively, for the SQL top level, use:

```
-sql=toplevel(proc1+proc2+proc3+...)
```

The following syntax is for a JPublisher `INPUT` file, where commas are used between stored procedure names:

```
SQL plsql_package (proc1, proc2, proc3, ...) AS ...
```

Notes:

- In an `INPUT` file, put a stored procedure name in quotes if it is case-sensitive. For example, "*`proc1`*". JPublisher assumes that names that are not in quotes are not case-sensitive.
- Case-sensitive names are not supported on the JPublisher command line.
- Specified stored procedure names can end in the wildcard character, "`%`". The specification "`myfunc%`", for example, matches all stored procedures that have their name starting with `myfunc`, such as `myfunc1`.

You can also specify the subset according to stored procedure names and argument types by using the following syntax:

```
myfunc(sqltype1, sqltype2, ...)
```

In this case, only those stored procedures that match in name and the number and types of arguments will be published. For example:

```
-sql=mypackage(myfunc1(NUMBER, CHAR)+myfunc2(VARCHAR2))
```

Declaration of SQL Statements to Translate

The `-sqlstatement` option enables you to publish `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements as Java methods. JPublisher generates SQLJ classes for this functionality.

The syntax of the `-sqlstatement` option is as follows:

```
-sqlstatement.class=ClassName:UserClassName#UserInterfaceName
-sqlstatement.methodName=sqlStatement
-sqlstatement.return={both|resultset|beans}
```

Use `-sqlstatement.class` to specify the Java class in which the method will be published. In addition to the JPublisher-generated class, you can optionally specify a user subclass of the generated class, a user interface for the generated class or subclass, or both. Functionality for subclasses and interfaces is the same as for the `-sql` option. If you also use the JPublisher `-package` option, then the class you specify will be in the specified package. The default class is `SQLStatements`.

Use `-sqlstatement.`*`methodName`* to specify the desired Java method name and the SQL statement. For a `SELECT` statement, use `-sqlstatement.return` to specify whether JPublisher should generate a method that returns a generic `java.sql.ResultSet` instance, a method that returns an array of JavaBeans, or both. *Generic* implies that the column types of the result set are unknown or unspecified.

For queries, however, the column types are actually known. This provides the option of returning specific results through an array of beans. The name of the method returning `ResultSet` will be *`methodName`*`()`. The name of the method returning JavaBeans will be *`methodName`*`Beans()`.

Note: If your desired class and interface names follow a pattern, then you can use the `-genpattern` option for convenience.

JPublisher INPUT file syntax is as follows:

```
SQLSTATEMENTS_TYPE ClassName AS UserClassName
                             IMPLEMENTS UserInterfaceName
SQLSTATEMENTS_METHOD aSqlStatement AS methodName
```

Here is a set of sample settings:

```
-sqlstatement.class=MySqlStatements
-sqlstatement.getEmp="select ename from emp
                      where ename=:{myname VARCHAR}"
-sqlstatement.return=both
```

These settings result in the generated code shown in "Generated Code: SQL Statement" on page A-7.

In addition, be aware that a style file specified through the -style option is relevant to the -sqlstatement option. If a SQL statement uses an Oracle data type X, which corresponds to a Java type Y, and type Y is mapped to a Java type Z in the style file, then methods generated as a result of the -sqlstatement option will use Z, and not Y.

For SELECT or DML statement results, you can use a style file to map the results to javax.xml.transform.Source, oracle.jdbc.rowset.OracleWebRowSet, or org.w3c.dom.Document.

> **See Also:** "JPublisher Styles and Style Files" on page 3-23 and "REF CURSOR Types and Result Sets Mapping" on page 3-7

Example: Using an XML Type This example shows the use of an XML type, SYS.XMLTYPE, with the -sqlstatement option. Assume the following table is created using SQL*Plus:

```
SQL>  CREATE TABLE xmltab (a XMLTYPE);
```

Now, assume the following JPublisher command to publish an INSERT statement:

```
% jpub  -u scott/tiger -style=webservices10
        -sqlstatement.addEle="insert into xmltab values(:{a sys.xmltype})"
```

This command causes the generation of the following methods:

```
public int addEle(javax.xml.transform.Source a) throws java.rmi.RemoteException;
public int addEleiS(javax.xml.transform.Source[] a)
                                        throws java.rmi.RemoteException;
```

This is because SYS.XMLTYPE is mapped to oracle.sql.SimpleXMLType, which the webservices10 style file further maps to javax.xml.transform.Source.

The method name, addEleiS, is used to avoid method overloading according to JPublisher naming conventions, with i reflecting the int return type and S reflecting the Source parameter type.

Note: This example assumes that JDK 1.4 is installed and used by JPublisher. If it is installed but not used by default, then you can set the -vm and -compiler-executable options to specify a JDK 1.4 JVM and compiler. For more information, refer to"Java Environment Options" on page 6-48.

Declaration of Object Types to Translate

The `-types` option lets you list one or more individual object types that you want JPublisher to translate. The syntax of the `-types` option is as follows:

```
-types=type_translation_syntax
```

> **Note:** The `-types` option is currently supported for compatibility, but it is deprecated. Use the `-sql` option instead.

You can use the `-types` option, for SQL object types only and when you do not need the generality of an `INPUT` file. Except for the fact that the `-types` option does not support PL/SQL packages, it is identical to the `-sql` option.

If you do not enter any types or packages to translate in the `INPUT` file or on the command line, then JPublisher translates all the types and packages in the schema to which you are connected. The command-line syntax lets you indicate three possible type translations.

- `-types=name_a`

  JPublisher interprets this syntax as:

  ```
  TYPE name_a
  ```

- `-types=name_a:name_b`

  JPublisher interprets this syntax as:

  ```
  TYPE name_a AS name_b
  ```

- `-types=name_a:name_b:name_c`

  JPublisher interprets this syntax as:

  ```
  TYPE name_a GENERATE name_b AS name_c
  ```

`TYPE`, `TYPE...AS`, and `TYPE...GENERATE...AS` commands have the same functionality as `SQL`, `SQL...AS`, and `SQL...GENERATE...AS` syntax.

See Also: "Understanding the Translation Statement" on page 6-52

Enter `-types=...` on the command line, followed by one or more object type translations that you want JPublisher to perform. If you enter more than one item, then the items must be separated by commas without any white space. For example, if you enter:

```
-types=CORPORATION,EMPLOYEE:OracleEmployee,ADDRESS:JAddress:MyAddress
```

JPublisher interprets this command as:

```
TYPE CORPORATION
TYPE EMPLOYEE AS OracleEmployee
TYPE ADDRESS GENERATE JAddress AS MyAddress
```

Connection Options

This section documents options related to the database connection that JPublisher uses. The options are discussed in the alphabetic order.

SQLJ Connection Context Classes

The `-context` option specifies the connection context class that JPublisher uses, and possibly declares, for SQLJ classes that JPublisher produces. The syntax of the `-context` option is as follows:

```
-context={generated|DefaultContext|user_defined}
```

The `-context=DefaultContext` setting is the default and results in any JPublisher-generated SQLJ classes using the SQLJ default connection context class, `sqlj.runtime.ref.DefaultContext`, for all connection contexts. This is sufficient for most uses.

Alternatively, you can specify any user-defined class that implements the standard `sqlj.runtime.ConnectionContext` interface and exists in the classpath. The specified class will be used for all connection contexts.

> **Note:** With a user-defined class, instances of that class must be used for output from the `getConnectionContext()` method or for input to the `setConnectionContext()` method. Refer to "Connection Contexts and Instances in SQLJ Classes" on page 4-10, for information about these methods.

The `-context=generated` setting results in an inner class declaration for the `_Ctx` connection context class in all SQLJ classes generated by JPublisher. So, each class uses its own SQLJ connection context class. This setting may be appropriate for Oracle8*i* compatibility mode, but it is otherwise not recommended. Using the `DefaultContext` class or a user-defined class avoids the generation of additional connection context classes. You can specify the `-context` option on the command line or in a properties file.

Notes for -context Usage in Backward-Compatibility Modes

If you use a backward-compatibility mode and use `.sqlj` files and the SQLJ translator directly, then a `-context=DefaultContext` setting gives you greater flexibility if you translate and compile your `.sqlj` files in separate steps, translating with the SQLJ `-compile=false` setting. If you are not using JDK 1.2-specific types, such as `java.sql.BLOB`, `CLOB`, `Struct`, `Ref`, or `Array`, then you can compile the resulting `.java` files under JDK 1.1, JDK 1.2, or later. This is *not* the case with the `-context=generated` setting, because SQLJ connection context classes in JDK 1.1 use `java.util.Dictionary` instances for object type maps, while SQLJ connection context classes in JDK 1.2 or later use `java.util.Map` instances.

A benefit of using the `-context=generated` setting, if you are directly manipulating `.sqlj` files, is that it permits full control over the way the SQLJ translator performs online checking. Specifically, you can check SQL user-defined types and PL/SQL packages against an appropriate exemplar database schema. However, because JPublisher generates `.sqlj` files from an existing schema, the generated code is already verified as correct through construction from that schema.

The Default datasource Option

You can use `-datasource` to specify the default data source for publishing SQL, PL/SQL, AQ, and server-side Java classes. With `-datasource` set, if the JDBC connection is not explicitly set by the application at run time, then the generated code will look up the specified Java Naming and Directory Interface (JNDI) location to get the data source and further get the JDBC connection from that data source.

The syntax of the -datasource option is as follows:

```
-datasource=jndi_location
```

JDBC Driver Class for Database Connection

The -driver option specifies the driver class that JPublisher uses for JDBC connections to the database. The syntax of this option is as follows:

```
-driver=driver_class_name
```

The default setting is:

```
-driver=oracle.jdbc.OracleDriver
```

This setting is appropriate for any Oracle JDBC driver.

Connection URL for Target Database

You can use the -url option to specify the URL of the database to which you want to connect. The syntax of the -url option is as follows:

```
-url=URL
```

The default setting is:

```
-url=jdbc:oracle:oci:@
```

To specify the JDBC Thin driver, use a setting of the following form:

```
-url=jdbc:oracle:thin:@host:port/servicename
```

In this syntax, *host* is the name of the host on which the database is running, *port* is the port number, and *servicename* is the name of the database service.

> **Note:** The use of system identifiers (SIDs) is deprecated in Oracle Database 10*g*, but it is still supported for backward compatibility. Their use is of the form *host*:*port*:*sid*.
>
> For the Oracle JDBC Oracle Call Interface (OCI) driver, use oci in the connect string in any new code. For backward compatibility, however, oci8 is still accepted for Oracle8*i* drivers.

User Name and Password for Database Connection

JPublisher requires the -user option, which specifies an Oracle user name and password, so that it can connect to the database. If you do not enter the -user option, then JPublisher prints an error message and stops execution.

The syntax of the -user option is as follows:

```
-user=username/password
-u username/password
```

Both formats are equivalent. The second one is provided as a convenient command-line shortcut.

For example, the following command directs JPublisher to connect to the database with the user name scott and password tiger:

```
% jpub -user=scott/tiger -input=demoin -dir=demo -mapping=oracle -package=corp
```

Options for Data Type Mappings

The following options control the data type mappings that JPublisher uses to translate object types, collection types, object reference types, and PL/SQL packages to Java classes:

- The `-usertypes` option controls JPublisher behavior for user-defined types, in conjunction with the `-compatible` option for `oracle` mapping. Specifically, it controls whether JPublisher implements the Oracle `ORAData` interface or the standard `SQLData` interface in generated classes, and whether JPublisher generates code for collection and object reference types.
- The `-numbertypes` option controls data type mappings for numeric types.
- The `-lobtypes` option controls data type mappings for the `BLOB`, `CLOB`, and `BFILE` types.
- The `-builtintypes` option controls data type mappings for non-numeric, non-LOB, and predefined SQL and PL/SQL types.

These four options are known as the type-mapping options.

For an object type, JPublisher applies the mappings specified by the type-mapping options to the object attributes and the arguments and results of any methods included with the object. The mappings control the types that the generated accessor methods support. For example, they support the types the `get`*`XXX`*`()` methods return and the `set`*`XXX`*`()` methods take.

For a PL/SQL package, JPublisher applies the mappings to the arguments and results of the methods in the package. For a collection type, JPublisher applies the mappings to the element type of the collection.

In addition, there is a subsection here for the `-style` option, which you can use to specify Java-to-Java type mappings, typically to support Web services. This involves an extra JPublisher step. A SQL type is mapped to a Java type that is not supported by Web services, in the JPublisher-generated base class. Then this Java type is mapped to a Java type that *is* supported by Web services, in the JPublisher-generated user subclass.

See Also: "JPublisher Styles and Style Files" on page 3-23

Mappings for Built-In Types

The `-builtintypes` option controls data type mappings for all the built-in data types except the LOB types, which are controlled by the `-lobtypes` option, and the different numeric types, which are controlled by the `-numbertypes` option. The syntax of the `-builtintypes` option is as follows:

```
-builtintypes={jdbc|oracle}
```

Table 6–2 lists the data types affected by the `-builtintypes` option and shows their Java type mappings for `-builtintypes=oracle` and `-builtintypes=jdbc`, which is the default.

Table 6–2 Mappings for Types Affected by the -builtintypes Option

SQL Data Type	Oracle Mapping Type	JDBC Mapping Type
`CHAR`, `CHARACTER`, `LONG`, `STRING`, `VARCHAR`, `VARCHAR2`	`oracle.sql.CHAR`	`java.lang.String`
`RAW`, `LONG RAW`	`oracle.sql.RAW`	`byte[]`

Table 6–2 (Cont.) Mappings for Types Affected by the -builtintypes Option

SQL Data Type	Oracle Mapping Type	JDBC Mapping Type
DATE	oracle.sql.DATE	java.sql.Timestamp
TIMESTAMP	oracle.sql.TIMESTAMP	java.sql.Timestamp
TIMESTAMP WITH TZ	oracle.sql.TIMESTAMPTZ	
TIMESTAMP WITH LOCAL TZ	oracle.sql.TIMESTAMPLTZ	

Mappings for LOB Types

The `-lobtypes` option controls data type mappings for LOB types. The syntax of the `-lobtypes` option is as follows:

```
-lobtypes={jdbc|oracle}
```

Table 6–3 shows how these types are mapped for `-lobtypes=oracle`, which is the default, and for `-lobtypes=jdbc`.

Table 6–3 Mappings for Types Affected by the -lobtypes Option

SQL Data Type	Oracle Mapping Type	JDBC Mapping Type
CLOB	oracle.sql.CLOB	java.sql.Clob
BLOB	oracle.sql.BLOB	java.sql.Blob
BFILE	oracle.sql.BFILE	oracle.sql.BFILE

Notes:

- `BFILE` is an Oracle-specific SQL type, so there is no standard `java.sql.Bfile` Java type.
- `NCLOB` is an Oracle-specific SQL type. It denotes an `NCHAR` form of use of a `CLOB` and is represented as an instance of `oracle.sql.NCLOB` in Java.
- The `java.sql.Clob` and `java.sql.Blob` interfaces were introduced in the JDK 1.2 versions.

Mappings for Numeric Types

The `-numbertypes` option controls data type mappings for numeric SQL and PL/SQL types. The syntax of the `-numbertypes` option is as follows:

```
-numbertypes={jdbc|objectjdbc|bigdecimal|oracle}
```

The following choices are available:

- In JDBC mapping, most numeric data types are mapped to Java primitive types, such as `int` and `float`, and `DECIMAL` and `NUMBER` are mapped to `java.math.BigDecimal`.
- In Object JDBC mapping, which is the default, most numeric data types are mapped to Java wrapper classes, such as `java.lang.Integer` and `java.lang.Float`. `DECIMAL` and `NUMBER` are mapped to `java.math.BigDecimal`.
- In `BigDecimal` mapping, all numeric data types are mapped to `java.math.BigDecimal`.

- In Oracle mapping, all numeric data types are mapped to `oracle.sql.NUMBER`.

Table 6–4 lists the data types affected by the `-numbertypes` option and shows their Java type mappings for `-numbertypes=jdbc` and `-numbertypes=objectjdbc`, which is the default.

Table 6–4 Mappings for Types Affected by the -numbertypes Option

SQL Data Type	JDBC Mapping Type	Object JDBC Mapping Type
`BINARY_INTEGER, INT, INTEGER, NATURAL, NATURALN, PLS_INTEGER, POSITIVE, POSITIVEN, SIGNTYPE`	`int`	`java.lang.Integer`
`SMALLINT`	`int`	`java.lang.Integer`
`REAL`	`float`	`java.lang.Float`
`DOUBLE PRECISION, FLOAT`	`double`	`java.lang.Double`
`DEC, DECIMAL, NUMBER, NUMERIC`	`java.math.BigDecimal`	`java.math.BigDecimal`

Mappings for User-Defined Types

The `-usertypes` option controls whether JPublisher implements the Oracle `ORAData` interface or the standard `SQLData` interface in generated classes for user-defined types. The syntax of the `-usertypes` option is as follows:

```
-usertypes={oracle|jdbc}
```

When `-usertypes=oracle`, which is the default, JPublisher generates `ORAData` classes for object, collection, and object reference types.

When `-usertypes=jdbc`, JPublisher generates `SQLData` classes for object types. JPublisher does not generate classes for collection or object reference types in this case. You must use `java.sql.Array` for all collection types and `java.sql.Ref` for all object reference types.

Notes:

- The `-usertypes=jdbc` setting requires JDK 1.2 or later, because the `SQLData` interface is a JDBC 2.0 feature.
- With certain settings of the `-compatible` option, a `-usertypes=oracle` setting results in classes that implement the deprecated `CustomDatum` interface instead of `ORAData`.

Mappings for All Types

The `-mapping` option specifies mapping for all data types, so offers little flexibility between types. The syntax of the `-mapping` option is as follows:

```
-mapping={jdbc|objectjdbc|bigdecimal|oracle}
```

Note: This option is deprecated in favor of the more specific type mapping options: `-usertypes`, `-numbertypes`, `-builtintypes`, and `-lobtypes`. However, it is still supported for backward compatibility.

The `-mapping=oracle` setting is equivalent to setting all the type mapping options to `oracle`. The other `-mapping` settings are equivalent to setting `-numbertypes` equal to the value of `-mapping` and setting the other type mapping options to their defaults. This is summarized in Table 6–5.

Table 6–5 Relation of -mapping Settings to Other Mapping Option Settings

-mapping Setting	-builtintypes=	-numbertypes=	-lobtypes=	-usertypes=
`-mapping=oracle`	`oracle`	`oracle`	`oracle`	`oracle`
`-mapping=jdbc`	`jdbc`	`jdbc`	`oracle`	`oracle`
`-mapping=objectjdbc` (default)	`jdbc`	`objectjdbc`	`oracle`	`oracle`
`-mapping=bigdecimal`	`jdbc`	`bigdecimal`	`oracle`	`oracle`

Note: Options are processed in the order in which they appear on the command line. Therefore, if the `-mapping` option precedes one of the specific type mapping options, `-builtintypes`, `-lobtypes`, `-numbertypes`, or `-usertypes`, then the specific type mapping option overrides the `-mapping` option for the relevant types. If the `-mapping` option follows one of the specific type mapping options, then the specific type mapping option is ignored.

Style File for Java-to-Java Type Mappings

JPublisher style files allow you to specify Java-to-Java type mappings. One use for this is to ensure that generated classes can be used in Web services. You use the `-style` option to specify the name of a style file. You can use the `-style` option multiple times. The settings accumulate in order. The syntax of the `-style` option is as follows:

```
-style=stylename
```

Typically, Oracle supplies the style files, but there may be situations in which you would edit or create your own. To use the Oracle style file for Web services in Oracle Database 10*g*, for example, use the following setting:

```
-style=webservices10
```

See Also: "JPublisher Styles and Style Files" on page 3-23

Type Map Options

JPublisher code generation is influenced by entries in the JPublisher user type map or default type map, primarily to make signatures with PL/SQL types accessible to JDBC. A type map entry has one of the following formats:

```
-type_map_option=opaque_sql_type:java_type
-type_map_option=numeric_indexed_by_table:java_numeric_type[max_length]
-type_map_option=char_indexed_by_table:java_char_type[max_length](elem_size)
-type_map_option=plsql_type:java_type:sql_type:sql_to_plsql_func:plsql_to_sql_func
```

In the type map syntax, *sql_to_plsql_func* and *plsql_to_sql_func* are for functions that convert between SQL and PL/SQL. Note that **[...]** and **(...)** are part of the syntax. Also note that some operating systems require you to quote command-line options that contain special characters.

The related options, which are discussed in alphabetic order in the following sections, are `-addtypemap`, `-adddefaulttypemap`, `-defaulttypemap`, and `-typemap`. The difference between `-addtypemap` and `-typemap` is that `-addtypemap` appends entries to the user type map, while `-typemap` replaces the existing user type map with the specified entries. Similarly, `-adddefaulttypemap` appends entries to the default type map, while `-defaulttypemap` replaces the existing default type map with the specified entries.

> **See Also:** "Type Mapping Support for OPAQUE Types" on page 3-11, "Type Mapping Support for Scalar Index-by Tables" on page 3-13, and "Type Mapping Support Through PL/SQL Conversion Functions" on page 3-16

Here are some sample type map settings, from a properties file that uses the `-defaulttypemap` and `-adddefaulttypemap` options:

```
jpub.defaulttypemap=SYS.XMLTYPE:oracle.xdb.XMLType
jpub.adddefaulttypemap=BOOLEAN:boolean:INTEGER:
SYS.SQLJUTL.INT2BOOL:SYS.SQLJUTL.BOOL2INT
jpub.adddefaulttypemap=INTERVAL DAY TO SECOND:String:CHAR:
SYS.SQLJUTL.CHAR2IDS:SYS.SQLJUTL.IDS2CHAR
jpub.adddefaulttypemap=INTERVAL YEAR TO MONTH:String:CHAR:
SYS.SQLJUTL.CHAR2IYM:SYS.SQLJUTL.IYM2CHAR
```

Be aware that you must avoid conflicts between the default type map and user type map.

Adding an Entry to the Default Type Map

Use the `-adddefaulttypemap` option to append an entry or a comma-delimited list of entries to the JPublisher default type map. In addition, JPublisher uses this option internally. The syntax of this option is:

```
-adddefaulttypemap=list_of_typemap_entries
```

Additional Entry to the User Type Map

Use the `-addtypemap` option to append an entry or a comma-delimited list of entries to the JPublisher user type map. The syntax of this option is:

```
-addtypemap=list_of_typemap_entries
```

Default Type Map for JPublisher

JPublisher uses the `-defaulttypemap` option internally to set up predefined type map entries in the default type map. The syntax of this option is:

```
-defaulttypemap=list_of_typemap_entries
```

The difference between the `-adddefaulttypemap` option and the `-defaulttypemap` option is that `-adddefaulttypemap` appends entries to the default type map, while `-defaulttypemap` replaces the existing default type map with the specified entries. To clear the default type map, use the following setting:

```
-defaulttypemap=
```

You may want to do this to avoid conflicts between the default type map and the user type map, for example.

> **See Also:** "JPublisher User Type Map and Default Type Map" on page 3-5 for additional information, including a caution about conflicts between the type maps.

Replacement of the JPublisher Type Map

Use the `-typemap` option to specify an entry or a comma-delimited list of entries to set up the user type map. The syntax of this option is:

```
-typemap=list_of_typemap_entries
```

The difference between the `-typemap` option and the `-addtypemap` option is that `-typemap` replaces the existing user type map with the specified entries and `-addtypemap` appends entries to the user type map. To clear the user type map, use the following setting.

```
-typemap=
```

You may want to do this to avoid conflicts between the default type map and the user type map, for example.

Java Code-Generation Options

This section documents options that specify JPublisher characteristics and behavior for Java code generation. For example, there are options to accomplish the following:

- Filter generated code according to parameter modes or parameter types
- Ensure that generated code conforms to the JavaBeans specification
- Specify naming patterns
- Specify how stubs are generated for user subclasses
- Specify whether generated code is serializable

The following options are described in alphabetical order: `-access`, `-case`, `-filtermodes`, `-filtertypes`, `-generatebean`, `-genpattern`, `-gensubclass`, `-methods`, `-omit_schema_names`, `-outarguments`, `-package`, `-serializable`, and `-tostring`.

Method Access

The `-access` option determines the access modifier that JPublisher includes in generated constructors, attribute setter and getter methods, member methods on object wrapper classes, and methods on PL/SQL packages. The syntax of this option is:

```
-access={public|protected|package}
```

JPublisher uses the possible option settings as follows:

- `public`

  Methods are generated with the `public` access modifier. This is the default option setting.

- `protected`

  Methods are generated with the `protected` access modifier.

- `package`

  The access modifier is omitted, so generated methods are local to the package.

You may want to use a setting of `-access=protected` or `-access=package` if you want to control the usage of the generated JPublisher wrapper classes. For example, when you provide customized versions of the wrapper classes as subclasses of the JPublisher-generated classes, but do not want to provide access to the generated superclasses.

You can specify the `-access` option on the command line or in a properties file.

> **Note:** Wrapper classes for object references and `VARRAY` and nested table types are not affected by the value of the `-access` option.

Case of Java Identifiers

For class or attribute names that you do not specify in an `INPUT` file or on the command line, the `-case` option affects the case of Java identifiers that JPublisher generates, including class names, method names, attribute names embedded within `get`*`XXX`*`()` and `set`*`XXX`*`()` method names, and arguments of generated method names. The syntax of this option is:

```
-case={mixed|same|lower|upper}
```

Table 6–6 describes the possible values for the `-case` option.

Table 6–6 Values for the -case Option

-case Option Value	Description
`mixed` (default)	The first letter of every word unit of a class name or of every word unit after the first word unit of a method name is in uppercase. All other characters are in lowercase. An underscore (_), a dollar sign ($), or any character illegal in Java constitutes a word unit boundary and is removed without warning. A word unit boundary also occurs after `get` or `set` in a method name.
`same`	JPublisher does not change the case of letters from the way they are represented in the database. Underscores and dollar signs are retained. JPublisher removes any other character illegal in Java and issues a warning message.
`upper`	JPublisher converts lowercase letters to uppercase and retains underscores and dollar signs. It removes any other character illegal in Java and issues a warning message.
`lower`	JPublisher converts uppercase letters to lowercase and retains underscores and dollar signs. It removes any other character illegal in Java and issues a warning message.

For class or attribute names that you specify through JPublisher options or the `INPUT` file, JPublisher retains the case of the letters in the specified name and overrides the `-case` option.

Method Filtering According to Parameter Modes

In some cases, particularly for generating code for Web services, not all parameter modes are supported in method signatures or attributes for the target usage of your code. The `-filtermodes` option enables you to filter generated code according to parameter modes. The syntax of this option is:

```
-filtermodes=list_of_modes_to_filter_out_or_filter_in
```

You can specify the following for the `-filtermodes` option:

- `in`
- `out`
- `inout`
- `return`

Start the option setting with a `1` to include all possibilities by default, which would mean no filtering. Then list specific modes or types each followed by a minus sign (`-`), indicating that the mode or type should be excluded. Alternatively, start with a `0` to include no possibilities by default, which would mean total filtering, then list specific modes or types each followed by a plus sign (`+`), indicating that the mode or type should be allowed.

The following examples would have the same result, allowing only methods that have parameters of the `in` or `return` mode. Separate the entries by commas.

```
-filtermodes=0,in+,return+

-filtermodes=1,out-,inout-
```

Method Filtering According to Parameter Types

In some cases, particularly for generating code for Web services, not all parameter types are supported in method signatures or attributes for the target usage of your code. The `-filtertypes` option enables you to filter generated code according to parameter types. The syntax of this option is:

```
-filtertypes=list_of_types_to_filter_out_or_filter_in
```

You can specify the following settings for the `-filtertypes` option:

- Any qualified Java type name

  Specify package and class, such as `java.sql.SQLData`, `oracle.sql.ORAData`.

- `.ORADATA`

  This setting indicates any `ORAData` or `SQLData` implementations.

- `.STRUCT`, `.ARRAY`, `.OPAQUE`, `.REF`

  Each of these settings indicates any types that implement `ORAData` or `SQLData` with the corresponding `_SQL_TYPECODE` specification.

- `.CURSOR`

  This setting indicates any SQLJ iterator types and `java.sql.ResultSet`.

- `.INDEXBY`

  This setting indicates any indexed-by table types.

- `.ORACLESQL`

  This setting indicates all `oracle.sql.`*`XXX`* types.

Start the option setting with a `1` to include all possibilities by default, indicating no filtering, then list specific modes or types each followed by a minus sign (`-`), indicating that the mode or type should be excluded. Alternatively, start with a `0` to include no possibilities by default, indicating total filtering, then list specific modes or types each followed by a plus sign (`+`), indicating that the mode or type should be allowed.

This first example filters out only .ORADATA and .ORACLESQL. The second example filters everything except .CURSOR and .INDEXBY:

```
-filtertypes=1,.ORADATA-,.ORACLESQL-

-filtertypes=0,.CURSOR+,.INDEXBY+
```

The .STRUCT, .ARRAY, .OPAQUE, and .REF settings are subcategories of the .ORADATA setting. Therefore, you can have specifications, such as the following, which filters out all ORAData and SQLData types except those with a typecode of STRUCT:

```
-filtertypes=1,.ORADATA-,.STRUCT+
```

Alternatively, to allow ORAData or SQLData types in general, with the exception of those with a typecode of ARRAY or REF:

```
-filtertypes=0,.ORADATA+,.ARRAY-,.REF-
```

Code Generation Adherence to the JavaBeans Specification

The -generatebean option is a flag that you can use to ensure that generated classes follow the JavaBeans specification. The syntax of this option is:

```
-generatebean={true|false}
```

The default setting is -generatebean=false. With the -generatebean=true setting, some generated methods are renamed so that they are not assumed to be JavaBean property getter or setter methods. This is accomplished by prefixing the method names with an underscore (_). For example, for classes generated from SQL table types, VARRAY, or indexed-by table, method names are changed as follows.

Method names are changed from:

```
public int getBaseType() throws SQLException;
public int getBaseTypeName() throws SQLException;
public int getDescriptor() throws SQLException;
```

to:

```
public int _getBaseType() throws SQLException;
public String _getBaseTypeName() throws SQLException;
public ArrayDecscriptor _getDescriptor() throws SQLException;
```

The changes in return types are necessary because the JavaBeans specification says that a getter method must return a bean property, but getBaseType(), getBaseTypeName(), and getDescriptor() do *not* return a bean property.

Class and Interface Naming Pattern

It is often desirable to follow a certain naming pattern for Java classes, user subclasses, and interfaces generated for user-defined SQL types or packages. The -genpattern option, which you can use in conjunction with the -sql or -sqlstatement option, enables you to define such patterns conveniently and generically. The syntax of this option is:

```
-genpattern=pattern_specifications
```

Consider the following explicit command-line options:

```
-sql=PERSON:PersonBase:PersonUser#Person
-sql=STUDENT:StudentBase:StudentUser#Student
```

```
-sql=GRAD_STUDENT:GradStudentBase:GradStudentUser#GradStudent
```

The following pair of options is equivalent to the preceding set of options:

```
-genpattern=%1Base:%1User#%1
-sql=PERSON,STUDENT,GRAD_STUDENT
```

By definition, `%1` refers to the default base names that JPublisher would create for each SQL type. By default, JPublisher would create the `Person` Java type for the `PERSON` SQL type, the `Student` Java type for the `STUDENT` SQL type, and the `GradStudent` Java type for the `GRAD_STUDENT` SQL type. So `%1Base` becomes `PersonBase`, `StudentBase`, and `GradStudentBase`, respectively. Similar results are produced for `%1User`.

If the `-sql` option specifies the output names, then `%2`, by definition, refers to the specified names. For example, the following pair of options has the same effect as the earlier pair:

```
-genpattern=%2Base:%2User#%2
-sql=PERSON:Person,STUDENT:Student,GRAD_STUDENT:GradStudent
```

> **Note:** This is the pattern expected for Web services. Specify an output name and use that as the interface name, and append `Base` for the generated class and `User` for the user subclass.

The following example combines the `-genpattern` option with the `-sqlstatement` option:

```
-sqlstatement.class=SqlStmts -genpattern=%2Base:%2User:%2
```

These settings are equivalent to the following:

```
-sqlstatement.class=SqlStmtsBase:SqlStmtsUser#SqlStmts
```

Generation of User Subclasses

The value of the `-gensubclass` option determines whether JPublisher generates initial source files for user-provided subclasses and, if so, what format these subclasses should have. The syntax of this option is:

```
-gensubclass={true|false|force|call-super}
```

For `-gensubclass=true`, which is the default, JPublisher generates code for the subclass only if it finds that no source file is present for the user subclass. The `-gensubclass=false` setting results in JPublisher not generating any code for user subclasses.

For `-gensubclass=force`, JPublisher always generates code for user subclasses. It overwrites any existing content in the corresponding `.java` and `.class` files if they already exist. Use this setting with caution.

The setting `-gensubclass=call-super` is equivalent to `-gensubclass=true`, except that JPublisher generates slightly different code. By default, JPublisher generates only constructors and methods necessary for implementing an interface, for example, the `ORAData` interface. JPublisher indicates how superclass methods or attribute setter and getter methods can be called, but places this code inside comments. With the `call-super` setting, all getters, setters, and other methods are generated.

The idea is that you can specify this setting if you use Java development tools based on class introspection. Only methods relating to SQL object attributes and SQL object methods are of interest, and JPublisher implementation details remain hidden. In this case you can point the tool at the generated user subclass.

You can specify the `-gensubclass` option on the command line or in a properties file.

Generation of Package Classes and Wrapper Methods

The `-methods` option determines whether:

- JPublisher generates wrapper methods for methods, or stored procedures in SQL object types and PL/SQL packages.
- Overloaded method names are allowed.
- Methods will attempt to reestablish a JDBC connection if an `SQLException` is caught.

The syntax for the `-methods` option is:

```
-methods={all|none|named|always,overload|unique,noretry|retry}
```

For `-methods=all`, which is the default setting among the first group of settings, JPublisher generates wrapper methods for all the methods in the SQL object types and PL/SQL packages it processes. This results in generation of a SQLJ class if the underlying SQL object or package actually defines methods and if not, a non-SQLJ class. Prior to Oracle Database 10*g*, SQLJ classes were always generated for the `all` setting.

For `-methods=none`, JPublisher does not generate wrapper methods. In this case, JPublisher does not generate classes for PL/SQL packages, because they would not be useful without wrapper methods.

For `-methods=named`, JPublisher generates wrapper methods only for the methods explicitly named in the `INPUT` file.

The `-methods=always` setting also results in wrapper methods being generated. However, for backward compatibility with the Oracle8*i* and Oracle9*i* JPublisher versions, this setting always results in SQLJ classes being generated for all SQL object types, regardless of whether the types define methods.

> **Note:** For backward compatibility, JPublisher also supports the setting `true` as equivalent to `all`, the setting `false` as equivalent to `none`, and the setting `some` as equivalent to `named`.

Among the `overload` and `unique` settings, `-methods=overload` is the default and specifies that method names in the generated code can be overloaded, such as the following:

```
int foo(int);
int foo(String);
```

Alternatively, the `-methods=unique` setting specifies that all method names must be unique. This is required for Web services. Consider the following functions:

```
function foo (a VARCHAR2(40)) return VARCHAR2;
function foo ( x int, y int) return int;
```

With the default `-methods=overload` setting, these functions are published as follows:

```
String foo(String a);
java.math.BigDecimal foo(java.math.BigDecimal x, java.math.BigDecimal y);
```

With the `-methods=unique` setting, these functions are published using a method-renaming mechanism based on the first letter of the return type and argument types, as shown in the following example:

```
String foo(String a);
java.math.BigDecimal fooBBB(java.math.BigDecimal x, java.math.BigDecimal y);
```

See Also: "Translation of Overloaded Methods" on page 4-6

With the `-methods=retry` setting, JPublisher generates constructors with `DataSource` arguments and extra code for each method published. A JDBC operation in a method is enclosed within a `try...catch` block. If an `SQLException` is raised when the method is processed, then the extra code will attempt to reestablish the JDBC connection and process the SQL operation again. If the attempt to reconnect fails, then the original `SQLException` is thrown again.

For `-methods=retry`, JPublisher generates code different from that generated for `-methods=noretry`, in two respects:

- An additional constructor, which takes a `DataSource` object as parameter, is generated. The `DataSource` object is used to get a new connection at operation invocation time.
- A new JDBC connection is requested if an `SQLException` is thrown.

The `-methods=retry` setting takes effect only for PL/SQL stored procedures, SQL statements, AQ, and Web services call-ins for Java classes.

Note: The use of `oracle.jdbc.pool.OracleDataSource` requires JDK 1.3 or later.

To specify a setting of `all`, `none`, `named`, or `always` at the same time as you specify a setting of `overload` or `unique` and a setting for `retry` or `noretry`, use a comma to separate the settings. This is shown in the following example:

```
-methods=always,unique,retry
```

You can specify the `-methods` option on the command line or in a properties file.

Omission of Schema Name from Name References

In publishing user-defined SQL types, such as objects and collections, when JPublisher references the type names in Java wrapper classes, it generally qualifies the type names with the database schema name, such as `SCOTT.EMPLOYEE` for the `EMPLOYEE` type in the `SCOTT` schema.

However, by specifying the `-omit_schema_names` option, you instruct JPublisher *not* to qualify SQL type names with schema names. The syntax of this option is:

```
-omit_schema_names
```

When this option is specified, names are qualified with a schema name only under the following circumstances:

- You declare the user-defined SQL type in a schema other than the one to which JPublisher is connected. A type from another schema always requires a schema name to identify it.
- You declare the user-defined SQL type with a schema name on the command line or in an `INPUT` file. The use of a schema name with the type name on the command line or `INPUT` file overrides the `-omit_schema_names` option.

Omitting the schema name makes it possible for you to use classes generated by JPublisher when you connect to a schema other than the one used when JPublisher is invoked, as long as the SQL types that you use are declared identically in the two schemas.

`ORAData` and `SQLData` classes generated by JPublisher include a `static final String` field that names the user-defined SQL type matching the generated class. When the code generated by JPublisher is processed, the SQL type name in the generated code is used to locate the SQL type in the database. If the SQL type name does not include the schema name, then the type is looked up in the schema associated with the current connection when the code generated by JPublisher is processed. If the SQL type name includes the schema name, then the type is looked up in that schema.

When the `-omit_schema_names` option is enabled, JPublisher generates the following code in the Java wrapper class for a SQL object type and similar code to wrap a collection type:

```
public Datum toDatum(Connection c) throws SQLException
   {
     if (__schemaName != null)
     {
       return _struct.toDatum(c, __schemaName + "." + _SQL_NAME);
     }
     return _struct.toDatum(c, typeName);
   }
   private String __schemaName = null;
   public void __setSchemaName(String schemaName) { __schemaName = schemaName; }
 }
```

The `__setSchemaName()` method enables you to explicitly set the schema name at run time so that SQL type names can be qualified by schema even if JPublisher was run with the `-omit_schema_names` option enabled. Being qualified by schema is necessary if a SQL type needs to be accessed from another schema.

Note: Although this option behaves as a boolean option, you cannot specify `-omit_schema_names=true` or `-omit_schema_names=false`. Specify `-omit_schema_names` to enable it, and do nothing to leave it disabled.

Holder Types for Output Arguments

There are no `OUT` or `IN OUT` designations in Java, but values can be returned through holders. In JPublisher, you can specify one of three alternatives for holders:

- Arrays, which is the default
- JAX-RPC holder types
- Function returns

The -outarguments option enables you to specify the mechanism to use, through a setting of array, holder, or return, respectively. This feature is particularly useful for Web services. The syntax of this option is:

```
-outarguments={array|holder|return}
```

See Also: "Treatment of Output Parameters" on page 4-1

Name for Generated Java Package

The -package option specifies the name of the Java package that JPublisher generates. The name appears in a package declaration in each generated class. The syntax for this option is:

```
-package=package_name
```

If you use the -dir and -d options, the directory structure in which JPublisher places the generated files reflects the package name as well as the -dir and -d settings.

Notes:

- If there are conflicting package settings between a -package option setting and a package setting in the INPUT file, the precedence depends on the order in which the -input and -package options appear on the command line. The -package setting takes precedence if that option is after the -input option. Otherwise, the INPUT file setting takes precedence.
- If you do not use the -dir and -d options, or if you explicitly give them empty settings, then JPublisher places all generated files directly in the current directory, with no package hierarchy, regardless of the -package setting.

See Also: "Output Directories for Generated Source and Class Files" on page 6-40

Example 1 Consider the following command:

```
% jpub -dir=/a/b -d=/a/b -package=c.d -sql=PERSON:Person ...
```

JPublisher generates the files /a/b/c/d/Person.java and /a/b/c/d/Person.class.

In addition, the Person class includes the following package declaration:

```
package c.d;
```

Example 2 Now consider the following command:

```
% jpub -dir=/a/b -d=/a/b -package=c.d -sql=PERSON:Person -input=myinputfile
```

Assume that myinputfile includes the following:

```
SQL PERSON AS e.f.Person
```

In this case, the package information in the INPUT file overrides the -package option on the command line. JPublisher generates the files /a/b/e/f/Person.java and /a/b/e/f/Person.class, with the Person class including the following package declaration:

```
package e.f;
```

If you do not specify a package name, then JPublisher does not generate any package declaration. The output `.java` files are placed directly into the directory specified by the `-dir` option or into the current directory by default. The output `.class` files are placed directly into the directory specified by the `-d` option or into the current directory.

Sometimes JPublisher translates a type that you do not explicitly request, because the type is required by another type that is translated. For example, it may be an attribute of the requested type. In this case, the `.java` and `.class` files declaring the required type are also placed into the package specified on the command line, in a properties file or the `INPUT` file.

By contrast, JPublisher never translates packages or stored procedures that you do not explicitly request, because packages or stored procedures are never strictly required by SQL types or by other packages or stored procedures.

Serializability of Generated Object Wrapper Classes

The `-serializable` option specifies whether the Java classes that JPublisher generates for SQL object types implement the `java.io.Serializable` interface. The default setting is `-serializable=false`. The syntax for this option is:

```
-serializable={true|false}
```

Please note the following if you choose to set `-serializable=true`:

- Not all object attributes are serializable. In particular, none of the Oracle LOB types, such as `oracle.sql.BLOB`, `oracle.sql.CLOB`, or `oracle.sql.BFILE`, can be serialized. Whenever you serialize objects with such attributes, the corresponding attribute values are initialized to `null` after deserialization.
- If you use object attributes of type `java.sql.Blob` or `java.sql.Clob`, then the code generated by JPublisher requires that the Oracle JDBC rowset implementation be available in the classpath. This is provided in the `ocrs12.jar` library at *ORACLE_HOME*`/jdbc/lib`. In this case, the underlying value of `Clob` and `Blob` objects is materialized, serialized, and subsequently retrieved.
- Whenever you deserialize objects containing attributes that are object references, the underlying connection is lost, and you cannot issue `setValue()` or `getValue()` calls on the reference. For this reason, JPublisher generates the following method into your Java classes whenever you specify `-serializable=true`:

  ```
  void restoreConnection(Connection)
  ```

  After deserialization, call this method once for a given object or object reference to restore the current connection into the reference or, respectively, into all transitively embedded references.

Generation of toString() Method on Object Wrapper Classes

You can use the `-tostring` flag to tell JPublisher to generate an additional `toString()` method for printing out an object value. The output resembles SQL code you would use to construct the object. The default setting is `false`. The syntax for this option is:

```
-tostring={true|false}
```

Rename main Method

You can use `-nomain=true` to avoid generating Java methods with the signature `main(String[])`. This option applies to SQL publishing and server-side Java class publishing. The syntax for this option is:

```
-nomain[=true|false]
```

The `-dbjava` option automatically sets `-nomain=true` because of Java stored procedure limitation. In case a method with the signature `main(String[])` is to be generated with the `-nomain=true` setting, then JPublisher will rename the method, for example, into `main0(Stringp[])`.

The default setting is:

```
-nomain=false
```

PL/SQL Code Generation Options

This section documents the following options that specify JPublisher behavior in generating PL/SQL code:

- `-overwritedbtypes`

  Specifies whether naming conflicts are checked before creating SQL types.

- `-plsqlfile`

  Specifies scripts to use in creating and dropping SQL types and PL/SQL packages.

- `-plsqlmap`

  Specifies whether PL/SQL wrapper functions are generated

- `-plsqlpackage`

  Specifies the name of the PL/SQL package in which JPublisher generates PL/SQL call specs, conversion functions, wrapper functions, and table functions.

These options are mostly used to support Java calls to stored procedures that use PL/SQL types. The options specify the creation and use of corresponding SQL types and the creation and use of PL/SQL conversion functions and PL/SQL wrapper functions that use the corresponding SQL types for input or output. This enables access through JDBC.

Generation of SQL types

JPublisher may generate new SQL types when publishing PL/SQL types and generating PL/SQL wrappers for server-side Java classes. The `-overwritedbtypes` option determines how JPublisher names the generated SQL types. The syntax for this option is:

```
-overwritedbtypes={true|false}
```

Prior to Oracle Database 10*g* release 2 (10.2), JPublisher checked the database for naming conflicts and chose a name, which was not already in use, for the generated SQL type. In Oracle Database 10*g* release 2 (10.2), JPublisher generates SQL type names by default, regardless of the existing type names in the database. The `-overwritedbtypes=true` setting, which is the default, overwrites the existing types if the type name is the same as that of the generated SQL type. This enables JPublisher to generate exactly the same PL/SQL wrappers over different runs.

To ensure that JPublisher does not overwrite any type inside the database while executing the generated PL/SQL wrapper, you must explicitly specify `-overwritedbtypes=false`.

A frequently reported problem in releases prior to Oracle Database 10*g* release 2 (10.2) is that after the generated PL/SQL wrapper is processed, rerunning the JPublisher command generates a different set of SQL types. A workaround for this problem is to run the PL/SQL dropper script before the JPublisher command is rerun.

File Names for PL/SQL Scripts

The `-plsqlfile` option specifies the name of a wrapper script and a dropper script generated by JPublisher. The syntax for this option is:

```
-plsqlfile=plsql_wrapper_script,plsql_dropper_script
```

The wrapper script contains instructions to create SQL types to map to PL/SQL types and instructions to create the PL/SQL package that JPublisher uses for any PL/SQL wrappers or call specs, conversion functions, wrapper functions, and table functions. The dropper script contains instructions to drop these entities.

You must load the generated files into the database, using SQL*Plus, for example, and run the wrapper script to install the types and package in the database.

If the files already exist, then they are overwritten. If no file names are specified, then JPublisher writes to files named `plsql_wrapper.sql` and `plsql_dropper.sql`.

JPublisher writes a note about the generated scripts, such as the following:

```
J2T-138, NOTE: Wrote PL/SQL package JPUB_PLSQL_WRAPPER to
file plsql_wrapper.sql. Wrote the dropping script to file plsql_dropper.sql.
```

Generation of PL/SQL Wrapper Functions

The `-plsqlmap` option specifies whether JPublisher generates wrapper functions for stored procedures that use PL/SQL types. Each wrapper function calls the corresponding stored procedure and invokes the appropriate PL/SQL conversion functions for PL/SQL input or output of the stored procedure. Only the corresponding SQL types are exposed to Java. The syntax for this option is:

```
-plsqlmap={true|false|always}
```

The setting can be any of the following:

- `true`

  This is the default. JPublisher generates PL/SQL wrapper functions only as needed. For any given stored procedure, if the Java code to call it and convert its PL/SQL types directly is simple enough and the PL/SQL types are used only as `IN` parameters or for the function return, then the generated code calls the stored procedure directly. It processes the PL/SQL input or output through the appropriate conversion functions.

- `false`

  JPublisher does not generate PL/SQL wrapper functions. If it encounters a PL/SQL type in a signature that cannot be supported by direct call and conversion, then it skips generation of Java code for the particular stored procedure.

- `always`

JPublisher generates a PL/SQL wrapper function for every stored procedure that uses a PL/SQL type. This is useful for generating a proxy PL/SQL package that complements an original PL/SQL package, providing Java-accessible signatures for those functions or procedures that are inaccessible from Java in the original package.

> **See Also:** "Type Mapping Support Through PL/SQL Conversion Functions" on page 3-16 and "Direct Use of PL/SQL Conversion Functions Versus Use of Wrapper Functions" on page 3-22

Package for Generated PL/SQL Code

The `-plsqlpackage` option specifies the name of the PL/SQL package into which JPublisher places any generated PL/SQL code. This includes PL/SQL wrappers or call specifications, conversion functions to convert between PL/SQL and SQL types, wrapper functions to wrap stored procedures that use PL/SQL types, and table functions. The syntax for this option is:

```
-plsqlpackage=name_of_PLSQL_package
```

By default, JPublisher uses the package `JPUB_PLSQL_WRAPPER`.

> **Note:** You must create this package in the database by running the SQL script generated by JPublisher.

Package for PL/SQL Index-By Tables

Use `-plsqlindextable=array` or `-plsqlindextable=int` to specify that PL/SQL index-by table of numeric and character types be mapped to Java array. The syntax for this option is:

```
-plsqlindextable=array|custom|int
```

The *int* specification defines the capacity of the Java array. The default capacity is 32768. The `-plsqlindextable=custom` setting specifies that PL/SQL index-by table be mapped to custom JDBC types, such as a class implementing `ORAData`.

The default setting is:

```
-plsqlindextable=custom
```

Input/Output Options

This section documents options related to JPublisher input and output files and locations. These are listed in the order in which they are discussed:

- `-compile`

  Use this option if you want to suppress compilation, and optionally, SQLJ translation, if JPublisher is in a backward-compatibility mode.

- `-dir`

  Use this option to specify where the generated source files are placed.

- `-d`

  Use this option to specify where the compiled class files are placed.

- `-encoding`

Use this option to specify the Java character encoding of the `INPUT` file that JPublisher reads and the `.sqlj` and `.java` files that JPublisher writes.

No Compilation or Translation

Use the `-compile` option to suppress the compilation of the generated `.java` files and, for backward-compatibility modes, to optionally suppress the translation of generated `.sqlj` files. The syntax for this option is:

```
-compile={true|false|notranslate}
```

With the default `true` setting, all generated classes are compiled into `.class` files. If you are in a backward-compatibility mode, then you can use the `-compile=notranslate` setting to suppress SQLJ translation and Java compilation of generated source files. This leaves you with `.sqlj` output from JPublisher, which you can translate and compile manually by using either the JPublisher `-sqlj` option or the SQLJ command-line utility directly. You can also use the `-compile=false` setting to proceed with SQLJ translation, but skip Java compilation. This leaves you with `.java` output from JPublisher, which you can compile manually.

If you are not in a backward-compatibility mode, such as if you use the default `-compatible=oradata` setting, then you can use a setting of `-compile=false` to skip compilation. In this scenario, the `notranslate` setting is not supported, because visible `.sqlj` files are not produced if you are not in a backward-compatibility mode.

> **See Also:** "Backward Compatibility Option" on page 6-46 and "Option to Access SQLJ Functionality" on page 6-45

Output Directories for Generated Source and Class Files

Use the `-dir` option to specify the root of the directory tree within which JPublisher places the `.java` source files or the `.sqlj` source files for backward-compatibility modes. The syntax for this option is:

```
-dir=directory_path
-d=directory_path
```

A setting of a period (`.`) explicitly specifies the current directory as the root of the directory tree. Similarly, use the `-d` option to specify the root of the directory tree within which JPublisher places compiled `.class` files, with the same functionality for a period (`.`) setting.

For each option with any nonempty setting, JPublisher also uses package information from the `-package` option or any package name included in an `SQL` option setting in the `INPUT` file. This information is used to determine the complete directory hierarchy for generated files.

> **See Also:** "Name for Generated Java Package" on page 6-35

For example, consider the following JPublisher command:

```
% jpub -user=scott/tiger -d=myclasses -dir=mysource -package=a.b.c
       -sql=PERSON:Person,STUDENT:Student
```

This results in the following output, relative to the current directory:

```
mysource/a/b/c/Person.java
mysource/a/b/c/PersonRef.java
mysource/a/b/c/Student.java
mysource/a/b/c/StudentRef.java
```

```
myclasses/a/b/c/Person.class
myclasses/a/b/c/PersonRef.class
myclasses/a/b/c/Student.class
myclasses/a/b/c/StudentRef.class
```

By default, source and class files are placed directly into the current directory, with no package hierarchy, regardless of the `-package` setting or any package specification in the `INPUT` file.

You can also explicitly specify this behavior with empty settings:

```
%jpub ... -d= -dir=
```

You can set these options on the command line or in a properties file.

> **Note:** SQLJ has `-dir` and `-d` options as well, with the same functionality. However, when you use the JPublisher `-sqlj` option to specify SQLJ settings, use the JPublisher `-dir` and `-d` options, which take precedence over any SQLJ `-dir` and `-d` settings.

Java Character Encoding

The `-encoding` option specifies the Java character encoding of the `INPUT` file that JPublisher reads and the source files that JPublisher writes. The default encoding is the value of the `file.encoding` system property or `8859_1` (ISO Latin-1), if this property is not set. The syntax for this option is:

```
-encoding=name_of_character_encoding
```

As a general rule, you do not have to set this option unless you specify an encoding for the SQLJ translator and Java compiler, which you can do with a SQLJ `-encoding` setting through the JPublisher `-sqlj` option. In this scenario, you should specify the same encoding for JPublisher as for SQLJ and the compiler.

You can use the `-encoding` option to specify any character encoding supported by your Java environment. If you are using the Sun Microsystems JDK, these options are listed in the `native2ascii` documentation, which you can find at the following URL:

http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/native2ascii.html

> **Note:** Encoding settings, either set through the JPublisher `-encoding` option or the Java `file.encoding` setting, do not apply to Java properties files, including those specified through the JPublisher `-props` option. Properties files always use the `8859_1` encoding. This is a feature of Java in general and not JPublisher in particular. However, you can use Unicode escape sequences in a properties file.

Options to Facilitate Web Services Call-Outs

This section documents options and related concepts for accessing Java classes from server-side Java or PL/SQL. In particular, these options may be used to access Web services client code from inside the database, referred to as Web services call-outs. This section comprises the following topics:

- WSDL Document for Java and PL/SQL Wrapper Generation
- Web Services Endpoint
- Proxy URL for WSDL
- Superuser for Permissions to Run Client Proxies

The following list is a summary of the options relevant to Web services call-outs and how they relate to each other:

- `-proxyclasses=`*`class1,class2,...,classN`*

  This option specifies Java classes for which Java and PL/SQL wrappers will be generated. For Web services, this option is used behind the scenes by the `-proxywsdl` option and is set automatically to process generated client proxy classes.

  Alternatively, you can use this option directly, for general purposes, when you want to create Java and PL/SQL wrappers for Java classes.

  The `-proxyclasses` option takes the `-proxyopts` setting as input.

- `-proxyopts=`*`setting1,setting2,...`*

  This option specifies JPublisher behavior in generating wrapper classes and PL/SQL wrappers. This is usually, but not necessarily, for Web services. For typical usage of the `-proxywsdl` option, the `-proxyopts` default setting is sufficient. If you directly use the `-proxyclasses` option, then you may want specific `-proxyopts` settings.

- `-proxywsdl=`*`WSDL_URL`*

  Use this option to generate Web services client proxy classes and appropriate Java and PL/SQL wrappers, given the WSDL document at the specified URL.

  The `-proxywsdl` option uses the `-proxyclasses` option behind the scenes and takes the `-proxyopts` setting as input.

- `-endpoint=`*`Web_services_endpoint`*

  Use this option in conjunction with the `-proxywsdl` option to specify the Web services endpoint.

- `-httpproxy=`*`proxy_URL`*

  Where the WSDL document is accessed through a firewall, use this option to specify a proxy URL to use in resolving the URL of the WSDL document.

- `-sysuser=`*`superuser_name/superuser_password`*

  Use this option to specify the name and password for the superuser account used to grant permissions for the client proxy classes to access Web services using HTTP.

Notes:

- The features described here require the `dbwsclient.jar` library to be installed in Oracle Database 10*g*.
- Several previously existing JPublisher options are used in conjunction with wrapper generation as discussed here: `-dir`, `-d`, `-plsqlmap`, `-plsqlfile`, `-plsqlpackage`, and `-package`. You can also specify a database connection through the `-user` and `-url` options so that JPublisher can load generated entities into the database.

WSDL Document for Java and PL/SQL Wrapper Generation

The syntax for the `-proxywsdl` option is:

```
-proxywsdl=WSDL_URL
```

This option is used as follows:

```
% jpub -proxywsdl=META-INF/HelloServiceEJB.wsdl ...
```

Given the Web services WSDL document at the specified URL, JPublisher directs the generation of Web services client proxy classes and generates appropriate Java and PL/SQL wrappers for Web services call-outs from the database. Classes to generate and process are determined from the WSDL document. JPublisher automatically sets the `-proxyclasses` option, uses the `-proxyopts` setting as input, and executes the following steps:

1. Invokes the Oracle Database Web services assembler tool to produce Web services client proxy classes based on the WSDL document. These classes use the Oracle Database Web services client run time to access the Web services specified in the WSDL document.
2. Creates Java wrapper classes for the Web services client proxy classes as appropriate or necessary. For each proxy class that has instance methods, a wrapper class is necessary to expose the instance methods as static methods. Even if there are no instance methods, a wrapper class is necessary if methods of the proxy class use anything other than Java primitive types in their calling sequences.
3. Creates PL/SQL wrappers for the generated classes, to make them accessible from PL/SQL. PL/SQL supports only static methods, so this step requires the wrapping of instance methods by static methods, This is performed in the previous step.
4. Loads generated code into the database assuming you have specified `-user` and `-url` settings and JPublisher has established a connection, unless you specifically bypass loading through the `-proxyopts=noload` setting.

Note: When using `-proxywsdl`:

- You must use the `-package` option to determine the package for generated Java classes.
- For `-proxyopts`, the default `jaxrpc` setting is sufficient for use with 10.0.x releases of Oracle Application Server 10*g*. This setting uses the singleton mechanism for publishing instance methods of the Web services client proxy classes. For use with the 9.0.4 release of Oracle Application Server 10*g* or with earlier releases, set `-proxyopts=soap`.

The -endpoint option is typically used in conjunction with the -proxywsdl option.

Web Services Endpoint

You can use the -endpoint option in conjunction with the -proxywsdl option to specify the Web services endpoint. The endpoint is the URL to which the Web service is deployed and from which the client accesses it. The syntax for this option is:

```
-endpoint=Web_services_endpoint
```

Use this option as follows:

```
% jpub -proxywsdl=META-INF/HelloServiceEJB.wsdl ...
       -endpoint=http://localhost:8888/javacallout/javacallout
```

With this command, the Java wrapper class generated by JPublisher includes the following code:

```
((Stub)m_port0)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
                "http://localhost:8888/javacallout/javacallout");
```

Without the -endpoint option, there would instead be the following commented code:

```
// Specify the endpoint and then uncomment the following statement:
// ((Stub)m_port0)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
//                 "<endpoint not provided>");
```

If you do not specify the endpoint in the JPublisher command line, then you must manually alter the generated wrapper class to uncomment this code and specify the appropriate endpoint.

Proxy URL for WSDL

If a WSDL document used for Web services call-outs is accessed through a firewall, use the -httpproxy option in conjunction with the -proxywsdl option to specify a proxy URL to use in resolving the URL of the WSDL document. The syntax for this option is:

```
-httpproxy=proxy_URL
```

For example:

```
% jpub ... -httpproxy=http://www-proxy.oracle.com:80
```

Superuser for Permissions to Run Client Proxies

Use the -sysuser option to specify the name and password of a superuser account. This account is used in running the JPublisher-generated PL/SQL script that grants permissions that allow client proxy classes to access Web services using HTTP. The syntax for this option is:

```
-sysuser=superuser_name/superuser_password
```

For example:

```
-sysuser=sys/change_on_install
```

Without a -sysuser setting, JPublisher does not load the generated script granting permissions. Instead, it asks you to execute the script separately.

If the -url setting specifies a thin driver, then you must set up a password file for SYS, which authorizes logon as SYS, through the thin driver. To set up a password file, you must:

1. Add the remote_login_passwordfile option to the database parameter file. You must use either of the following settings:

   ```
   remote_login_passwordfile=shared

   remote_login_passwordfile=exclusive
   ```

2. Create a password file, if you have not already created one. You can do this by running the following command, where *ORACLE_HOME*/dbs/ is an existing directory:

   ```
   orapwd file="ORACLE_HOME/dbs/orapwlsqlj1" password=change_on_install
   entries=100 force=y
   ```

3. Grant remote logon privileges to a user. This can be done as follows:

   ```
   % sqlplus /nolog
   SQL> CONN / AS sysdba
   Connected.
   SQL> GRANT sysdba TO scott;
   Grant succeeded.
   ```

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for details of setting up a remote SYS logon

Option to Access SQLJ Functionality

This section documents the -sqlj option, which you can use to pass SQLJ options to the SQLJ translator through the JPublisher command line.

Settings for the SQLJ Translator

In Oracle Database 10*g*, SQLJ translation is automatic by default when you run JPublisher. Translation is transparent, with no visible .sqlj files resulting from JPublisher code generation.

However, you can still specify SQLJ settings for the JPublisher invocation of the SQLJ translator by using the JPublisher -sqlj option. The syntax for this option is:

```
-sqlj=sqlj_options
```

For example:

```
% jpub -user=scott/tiger -sqlj -optcols=true -optparams=true
       -optparamdefaults=datatype1(size1),datatype2(size)
```

Notes:

- There is no equal sign (=) following -sqlj.
- All other JPublisher options must precede the -sqlj option. Any option setting following -sqlj is taken to be a SQLJ option and is passed to the SQLJ translator. In the preceding example, -optcols, -optparams, and -optparamdefaults are SQLJ options.

You can also run JPublisher solely to translate `.sqlj` files that have already been produced explicitly, such as if you run JPublisher with the `-compatible=sqlj` setting, which skips the automatic SQLJ translation step and results in `.sqlj` output files from JPublisher. In this case, use no JPublisher options other than `-sqlj`. This is a way to accomplish manual SQLJ translation if the `sqlj` front-end script or executable is unavailable.

The commands following `-sqlj` are equivalent to the command you would give to the SQLJ translator utility directly. Here is an example:

```
% jpub -sqlj -d=outclasses -warn=none -encoding=SJIS Foo.sqlj
```

This is equivalent to the following, if the SQLJ command-line translator is available:

```
% sqlj -d=outclasses -warn=none -encoding=SJIS Foo.sqlj
```

Notes:

- As an alternative to specifying SQLJ option settings through the `-sqlj` option, you can specify them in the `sqlj.properties` file that JPublisher supports.
- The `-compiler-executable` option, if set, is passed to the SQLJ translator to specify the Java compiler that the translator will use to compile Java code.

Backward Compatibility Option

This section documents the `-compatible` option, which you can use to specify any of the following:

- The interface for JPublisher to implement in generated classes
- Whether JPublisher should skip SQLJ translation, resulting in visible `.sqlj` output files
- A backward-compatibility mode to use JPublisher output in an Oracle9*i* or Oracle8*i* environment

See Also: "Backward Compatibility and Migration" on page 5-5

Backward-Compatible Oracle Mapping for User-Defined Types

The `-compatible` option has two modes of operation:

- Through a setting of `oradata` or `customdatum`, you can explicitly specify an interface to be implemented by JPublisher-generated custom Java classes.
- Through a setting of `sqlj`, `8i`, `both8i`, or `9i`, you can specify a backward-compatibility mode.

You can select either of the two modes, but not both.

The syntax for this option is:

```
-compatible={oradata|customdatum|both8i|8i|9i|10.1|sqlj}
```

Using -compatible to Specify an Interface

If `-usertypes=oracle`, then you have the option of setting `-compatible=customdatum`, to implement the deprecated `CustomDatum` interface

in your generated classes for user-defined types, instead of the default `ORAData` interface. `CustomDatum` was replaced by `ORAData` in Oracle9*i* Database, but is still supported for backward compatibility.

The default setting to use the `ORAData` interface is `oradata`. If you set `-usertypes=jdbc`, then a `-compatible` setting of `customdatum` or `oradata` is ignored.

If you use JPublisher in a pre-Oracle9*i* Database environment, in which the `ORAData` interface is not supported, then the `CustomDatum` interface is used automatically if `-usertypes=oracle`. You will receive an informational warning if `-compatible=oradata`, but the generation will take place.

Using -compatible to Specify a Backward-Compatibility Mode

Use the `sqlj`, `10.1`, `9i`, `8i`, or `both8i` setting to specify a backward-compatibility mode.

The `-compatible=sqlj` setting instructs JPublisher to skip SQLJ translation and instead produce `.sqlj` files that you can work with directly. The `sqlj` setting has no effect on the generated code itself. To translate the resulting `.sqlj` files, you can use the SQLJ translator directly, if available, or use the JPublisher `-sqlj` option.

See Also: "Option to Access SQLJ Functionality" on page 6-45

The `-compatibility=10.1` setting specifies Oracle Database 10*g* release 1 (10.1) compatibility mode. In this mode, the JPublisher option `-dbjava` behaves the same as `-java` in Oracle Database 10*g* release 1 (10.1).

The `-compatibility=9i` setting specifies Oracle9*i* compatibility mode. In this mode, JPublisher generates `.sqlj` files with the same code as would be generated by the Oracle9*i* version.

The `-compatible=8i` setting specifies Oracle8*i* compatibility mode. This mode uses the `CustomDatum` interface, generating `.sqlj` files with the same code that would be generated by Oracle8*i* versions of JPublisher. The `8i` setting is equivalent to setting several individual JPublisher options for backward compatibility to Oracle8*i*. For example, behavior of method generation is equivalent to that for `-methods=always`, and generation of connection context declarations is equivalent to that for `-context=generated`.

The `-compatible=both8i` setting is for an alternative Oracle8*i* compatibility mode. With this setting, wrapper classes are generated to implement both the `ORAData` and `CustomDatum` interfaces. Code is generated as it would have been by the Oracle8*i* version of JPublisher. This setting is generally preferred over the `-compatible=8i` setting, because support for `ORAData` is required for programs running in the middle tier, such as in Oracle Application Server. However, using `ORAData` requires an Oracle9*i* release 1 (9.0.1) or later JDBC driver.

Note: In any compatibility mode that results in the generation of visible `.sqlj` files, remember that if you are generating Java wrapper classes for a SQL type hierarchy and any of the types contains stored procedures, then, by default, JPublisher generates `.sqlj` files for all the SQL types, and not just the types that have stored procedures.

Java Environment Options

This section discusses JPublisher options that you can use to determine the Java environment:

- The `-classpath` option specifies the Java classpath that JPublisher and SQLJ use to resolve classes during translation and compilation.
- The `-compiler-executable` option specifies the Java compiler for compiling the code generated by JPublisher.
- The `-vm` option specifies the JVM through which JPublisher is invoked.

In a UNIX environment, the `jpub` script specifies the location of the Java executable that runs JPublisher. This script is generated at the time you install your database or application server instance. If the `jpub` script uses a Java version prior to JDK 1.4, then some JPublisher functionality for Web services, such as call-outs and mapping to the `SYS.XMLType`, are unavailable.

Classpath for Translation and Compilation

Use the `-classpath` option to specify the Java classpath for JPublisher to use in resolving Java source and classes during translation and compilation. The syntax for this option is:

```
-classpath=path1:path2:...:pathN
```

The following command shows an example of its usage, adding new paths to the existing classpath:

```
% jpub -user=scott/tiger -sql=PERSON:Person,STUDENT:Student
       -classpath=.:$ORACLE_HOME/jdbc/lib/ocrs12.jar:$CLASSPATH
```

> **Note:** SQLJ also has a `-classpath` option. If you use the SQLJ `-classpath` option, following the JPublisher `-sqlj` option, then that setting is used for the classpath for translation and compilation, and any JPublisher `-classpath` option setting is ignored. It is more straightforward to use only the JPublisher `-classpath` option.

Java Compiler

Use the `-compiler-executable` option if you want Java code generated by JPublisher to be compiled by anything other than the compiler that JPublisher would use by default on your system. Specify the path to an alternative compiler executable file. The syntax for this option is:

```
-compiler-executable=path_to_compiler_executable
```

Java Version

Use the `-vm` option if you want to use a JVM other than the JVM that JPublisher would use by default on your system. Specify the path to an alternative Java executable file. The syntax for this option is:

```
-vm=path_to_JVM_executable
```

As an example, assume that JDK 1.4 is installed on a UNIX system at the location `JDK14`, relative to the current directory. Run JPublisher with the following command

to use the JDK 1.4 JVM and compiler when publishing Web services client proxy classes:

```
% jpub -vm=JDK14/bin/java -compiler-executable=JDK14/bin/javac
       -proxywsdl=hello.wsdl
```

SQLJ Migration Options

In Oracle Database 10*g* release 2 (10.2), JPublisher provides the following command-line options to support migrating SQLJ to JDBC applications:

- `-migrate`

  This option enables you to turn on SQLJ migration. The syntax for setting this option is as follows:

  ```
  -migrate[=true|false]
  ```

  The `-migrate` and `-migrate=true` settings are equivalent. Both these settings indicate that JPublisher should migrate SQLJ programs specified on the command line to JDBC programs. The `-migrate=false` turns off the migration mode, and SQLJ programs are translated and compiled into a Java program that possibly depends on the SQLJ run time. By default, if this option is not specified, then JPublisher behaves like `-migrate=false`.

- `-migconn`

  This option enables you to specify the default JDBC connection used by the migrated code. The default JDBC connection replaces the default `DefaultContext` instance in the SQLJ run time. The syntax for setting this option is as follows:

  ```
  -migconn=getter:getter,setter:setter
  -migconn=name[[:datasource|modifier][,modifier]*]
  ```

  In the first syntax, the `getter` and `setter` settings specify the getter and setter methods for the default connection. For example:

  ```
  -migconn=getter:Test.getDefConn,setter:Test.setDefConn
  ```

  In the second syntax, the *`name`* setting specifies the default JDBC connection. The optional *`datasource`* setting provides a JNDI data source location for initializing the default JDBC connection. The *`modifier`* settings add the modifiers for the default connection variable. You can specify more than one modifier by using multiple *`modifier`* settings. Examples of the usage of the second syntax are as follows:

  ```
  -migconn=_defaultConn:public,static
  -migconn=Test._defaultConn
  -migconn=Test._defaultConn:jdbc/MyDataSource
  -migconn=_defaultConn:public,static,final
  ```

- `-migrsi`

  This option enables you to specify an interface for all `ResultSet` iterator classes. The syntax for setting this option is as follows:

  ```
  -migrsi=java_interface_name
  ```

  For example:

  ```
  -migrsi=ResultSetInterface
  ```

- `-migsync`

  This option enables you to mark static variables as `synchronized`. The syntax for setting this option is as follows:

  ```
  -migsync[=true|false]
  ```

  The `-migsync` and `-migsync=true` settings mark static variables generated for migration purpose as synchronized. If you do not want the variables to be marked as synchronized, then set `-migsync=false`. By default, JPublisher behaves like `-migsync=true`.

- `-migdriver`

  This option enables you to specify the JDBC driver registered by the migrated code. The syntax for setting this option is as follows:

  ```
  -migdriver=class_name|no
  ```

  For example:

  ```
  -migdriver=oracle.jdbc.driver.OracleDriver
  ```

  By default, JPublisher behaves like `-migdriver=oracle.jdbc.driver.OracleDriver`. If you do not want the driver registration code to be generated during migration, then set this option as follows:

  ```
  -migdriver=no
  ```

- `-migcodegen`

  This option enables you to specify whether the migrated code depends on an Oracle JDBC driver or a generic JDBC driver. The syntax for setting this option is as follows:

  ```
  -migcodegen=oracle|jdbc
  ```

  The default behavior is `-migcodegen=oracle`, indicating that Oracle-specific JDBC APIs are used in the migrated code.

- `-migserver`

  This option enables you to specify whether to migrate the program to be used on the server. The syntax for setting this option is as follows:

  ```
  -migserver
  ```

  See Also:

  `http://www.oracle.com/technology/tech/java/sqlj_jdbc/pdf/oracle_sqlj_roadmap.pdf`

JPublisher Input Files

The following sections describe the structure and contents of JPublisher input files:

- Properties File Structure and Syntax
- INPUT File Structure and Syntax
- INPUT File Precautions

Properties File Structure and Syntax

A properties file is an optional text file in which you can specify frequently used options. Specify the name of the properties file on the JPublisher command line with the `-props` option.

> **Note:** `-props` is the only option that you cannot specify in a properties file.

On each line in a properties file, enter only one option with its associated value. Enter each option setting with the following prefix, including the period:

```
jpub.
```

The `jpub.` prefix is case-sensitive. White space is permitted only directly in front of `jpub.`. Any other white space within the option line is significant.

Alternatively, JPublisher permits you to specify options with a double-dash (`--`), which is the syntax for SQL comments, as part of the prefix:

```
-- jpub.
```

A line that does not start with either of the prefixes shown is simply ignored by JPublisher.

In addition, you can use line continuation to spread a JPublisher option over several lines in the properties file. A line to be continued must have a backslash character (`\`) as the last character, immediately after the text of the line. Any leading space or double dash (`--`) on the line that follows the backslash is ignored. Consider the following sample entries:

```
/* The next three lines represent a JPublisher option
   jpub.sql=SQL_TYPE:JPubJavaType:MyJavaType,\
            OTHER_SQL_TYPE:OtherJPubType:MyOtherJavaType,\
            LAST_SQL_TYPE:My:LastType
*/
-- The next two lines represent another JPublisher option
-- jpub.addtypemap=PLSQL_TYPE:JavaType:SQL TYPE\
--                :SQL_TO_PLSQL_FUNCTION:PLSQL_TO_SQL_FUNCTION
```

Using this functionality, you can embed JPublisher options in SQL scripts, which may be useful when setting up PL/SQL-to-SQL type mappings.

JPublisher reads the options in the properties file in order, as if its contents were inserted on the command line at the point where the `-props` option is located. If you specify an option more than once, then the last value encountered by JPublisher overrides previous values, except for the following options, which are cumulative:

- `jpub.sql` (or the deprecated `jpub.types`)
- `jpub.java`
- `jpub.style`
- `jpub.addtypemap`
- `jpub.adddefaulttypemap`

For example, consider the following command:

```
% jpub -user=scott/tiger -sql=employee -mapping=oracle -case=lower -package=corp
-dir=demo
```

Now consider the following:

```
% jpub -props=my_properties
```

This command is equivalent to the first example if you assume that `my_properties` has a definition such as the following:

```
-- jpub.user=scott/tiger
// jpub.user=cannot_use/java_line_comments
jpub.sql=employee
/*
jpub.mapping=oracle
*/
Jpub.notreally=a jpub option
   jpub.case=lower
jpub.package=corp
    jpub.dir=demo
```

You must include the `jpub.` prefix at the beginning of each option name. If you enter anything other than white space or double dash (`--`) before the option name, then JPublisher ignores the entire line.

The preceding example illustrates that white space before `jpub.` is okay. It also shows that the `jpub.` prefix must be all lowercase, otherwise it is ignored. Therefore the following line from the preceding example will be ignored:

```
Jpub.notreally=a jpub option
```

INPUT File Structure and Syntax

Specify the name of the `INPUT` file on the JPublisher command line with the `-input` option. This file identifies SQL user-defined types and PL/SQL packages that JPublisher should translate. It also controls the naming of the generated classes and packages. Although you can use the `-sql` command-line option to specify user-defined types and packages, an `INPUT` file allows you a finer degree of control over how JPublisher translates them.

If you do not specify types or packages to translate in an `INPUT` file or on the command line, then JPublisher translates all user-defined types and PL/SQL packages in the schema to which it connects.

Understanding the Translation Statement

The translation statement in the `INPUT` file identifies the names of the user-defined types and PL/SQL packages that you want JPublisher to translate. Optionally, the translation statement can also specify a Java name for the type or package, a Java name for attribute identifiers, and whether there are any extended classes.

One or more translation statements can appear in the `INPUT` file. The structure of a translation statement is as follows:

```
( SQL name
| SQL [schema_name.]toplevel  [(name_list)]
| TYPE type_name)
[GENERATE java_name_1]
[AS java_name_2]
[TRANSLATE
     database_member_name AS simple_java_name
 { , database_member_name AS simple_java_name}*
```

```
]
```

The following sections describe the components of the translation statement.

SQL *name* | TYPE *type_name* Enter `SQL` *`name`* to identify a SQL type or a PL/SQL package that you want JPublisher to translate. JPublisher examines the *`name`*, determines whether it is for a user-defined type or a PL/SQL package, and processes it appropriately. If you use the reserved word `toplevel` in place of *`name`*, JPublisher translates the top-level subprograms in the schema to which JPublisher is connected.

Instead of `SQL`, it is permissible to enter `TYPE` *`type_name`* if you are specifying only object types. However, the `TYPE` syntax is deprecated.

You can enter *`name`* as *`schema_name`*.*`name`* to specify the schema to which the SQL type or package belongs. If you enter *`schema_name`*.`toplevel`, JPublisher translates the top-level subprograms in schema *`schema_name`*. In conjunction with `TOPLEVEL`, you can also supply *`name_list`*, which is a comma-delimited list of names to be published, enclosed in parentheses. JPublisher considers only top-level functions and procedures that match this list. If you do not specify this list, JPublisher generates code for all top-level subprograms.

Note: If a user-defined type is defined in a case-sensitive way in SQL, then you must specify the name in quotes. For example:

```
SQL "CaseSenstiveType" AS CaseSensitiveType
```

Alternatively, you can also specify a schema name that is not case-sensitive:

```
SQL SCOTT."CaseSensitiveType" AS CaseSensitiveType
```

You can also specify a case-sensitive schema name:

```
SQL "Scott"."CaseSensitiveType" AS CaseSensitiveType
```

The `AS` clause is optional.

Avoid situations where a period (".") is part of the schema name or the type name itself.

GENERATE *java_name_1* AS *java_name_2* The `AS` clause specifies the name of the Java class that represents the SQL user-defined type or PL/SQL package being translated.

When you use the `AS` clause without a `GENERATE` clause, JPublisher generates the class in the `AS` clause and maps it to the SQL type or PL/SQL package.

When you use both the `GENERATE` clause and the `AS` clause for a SQL user-defined type, the `GENERATE` clause specifies the name of the Java class that JPublisher generates, which is referred to as the base class. The `AS` clause specifies the name of a Java class that extends the generated base class, which is referred to as the user subclass. JPublisher produces an initial version of the user subclass, and you will typically add code for your desired functionality. JPublisher maps the SQL type to the user subclass, and not to the base class. If you later run the same JPublisher command to republish the SQL type, then the generated class is overwritten, but the user subclass is not.

The *`java_name_1`* and *`java_name_2`* can be any legal Java names and can include package identifiers. The case of the Java names overrides the `-case` option.

TRANSLATE *database_member_name* AS *simple_java_name* This clause optionally specifies a different name for an attribute or method. The *database_member_name* is the name of an attribute of a SQL object type or the name of a method of an object type or PL/SQL package. The attribute or method is to be translated to *simple_java_name*, which can be any legal Java name. The case of the Java name overrides the `-case` option. This name cannot have a package name.

If you do not use `TRANSLATE...AS` to rename an attribute or method, or if JPublisher translates an object type not listed in the `INPUT` file, then JPublisher uses the database name of the attribute or method as the Java name. If applicable, the Java name is modified according to the setting of the `-case` option. Reasons why you may want to rename an attribute or method include:

- The name contains characters other than letters, digits, and underscores.
- The name conflicts with a Java keyword.
- The type name conflicts with another name in the same scope. This can happen, for example, if the program uses two types with the same name from different schemas.

Remember that your attribute names will appear embedded within `get`*XXX*`()` and `set`*XXX*`()` method names, so you may want to capitalize the first letter of your attribute names. For example, if you enter:

```
TRANSLATE FIRSTNAME AS FirstName
```

JPublisher generates a `getFirstName()` method and a `setFirstName()` method. In contrast, if you enter:

```
TRANSLATE FIRSTNAME AS firstName
```

JPublisher generates a `getfirstName()` method and a `setfirstName()` method.

Note: The Java keyword `null` has special meaning when used as the target Java name for an attribute or method, such as in the following example:

```
TRANSLATE FIRSTNAME AS null
```

When you map a SQL method to `null`, JPublisher does not generate a corresponding Java method in the mapped Java class. When you map a SQL object attribute to `null`, JPublisher does not generate the getter and setter methods for the attribute in the mapped Java class.

Package Naming Rules in the INPUT File You can specify a package name by using a fully qualified class name in the `INPUT` file. If you use a simple, unqualified class name in the `INPUT` file, then the fully qualified class name includes the package name from the `-package` option. This is demonstrated in the following examples:

- Assume the following in the `INPUT` file:

  ```
  SQL A AS B
  ```

  And assume the setting `-package=a.b`. In this case, `a.b` is the package and `a.b.B` is the fully qualified class name.

- Assume that you enter the following in the `INPUT` file and there is no `-package` setting:

```
SQL A AS b.C
```

The package is `b`, and `b.C` is the fully qualified class name.

See Also: "Name for Generated Java Package" on page 6-35

Note: If there are conflicting package settings between a `-package` option setting and a package setting in the `INPUT` file, then the precedence depends on the order in which the `-input` and `-package` options appear on the command line. The `-package` setting takes precedence if that option is after the `-input` option, else the `INPUT` file setting takes precedence.

Translating Additional Types It may be necessary for JPublisher to translate additional types that are not listed in the `INPUT` file. This is because JPublisher analyzes the types in the `INPUT` file for dependencies before performing the translation and translates any additional required types.

Consider the example in "Sample JPublisher Translation" on page 1-22. Assume that the object type definition for `EMPLOYEE` includes an attribute called `ADDRESS`, and `ADDRESS` is an object with the following definition:

```
CREATE OR REPLACE TYPE address AS OBJECT
(
    street     VARCHAR2(50),
    city       VARCHAR2(50),
    state      VARCHAR2(30),
    zip        NUMBER
);
```

In this case, JPublisher would first translate `ADDRESS`, because that would be necessary to define the `EMPLOYEE` type. In addition, `ADDRESS` and its attributes would all be translated in the same case, because they are not specifically mentioned in the `INPUT` file. A class file would be generated for `Address.java`, which would be included in the package specified on the command line.

JPublisher does not translate PL/SQL packages you do not request, because user-defined types or other PL/SQL packages cannot have dependencies on PL/SQL packages.

Sample Translation Statement

To better illustrate the function of the `INPUT` file, consider an updated version of the example in "Sample JPublisher Translation" on page 1-22.

Consider the following command:

```
% jpub -user=scott/tiger -input=demoin -numbertypes=oracle -usertypes=oracle
-dir=demo -d=demo -package=corp -case=same
```

And assume that the `INPUT` file `demoin` contains the following:

```
SQL employee AS Employee
TRANSLATE NAME AS Name HIRE_DATE AS HireDate
```

The `-case=same` option specifies that generated Java identifiers maintain the same case as in the database, except where you specify otherwise. Any identifier in a `CREATE TYPE` or `CREATE PACKAGE` declaration is stored in uppercase in the database

unless it is quoted. In this example, the `-case` option does not apply to the `EMPLOYEE` type, because `EMPLOYEE` is specified to be translated as the Java class `Employee`.

For attributes, attribute identifiers not specifically mentioned in the `INPUT` file remain in uppercase, but JPublisher translates `NAME` and `HIRE_DATE` as `Name` and `HireDate`, as specified.

The translated `EMPLOYEE` type is written to the following files, relative to the current directory, for UNIX in this example, reflecting the `-package`, `-dir`, and `-d` settings:

```
demo/corp/Employee.java
demo/corp/Employee.class
```

INPUT File Precautions

This section describes possible `INPUT` file error conditions that JPublisher will currently *not* report. There is also a section for reserved terms.

Requesting the Same Java Class Name for Different Object Types

If you request the same Java class name for two different object types, the second class overwrites the first without warning. Consider that the `INPUT` file contains:

```
type PERSON1 as Person
type PERSON2 as Person
```

JPublisher creates the `Person.java` file for `PERSON1` and then overwrites it for the `PERSON2` type.

Requesting the Same Attribute Name for Different Object Attributes

If you request the same attribute name for two different object attributes, JPublisher generates `get`*`XXX`*`()` and `set`*`XXX`*`()` methods for both attributes without issuing a warning message. The question of whether the generated class is valid in Java depends on whether the two `get`*`XXX`*`()` methods with the same name and the two `set`*`XXX`*`()` methods with the same name have different argument types so that they may be unambiguously overloaded.

Specifying Nonexistent Attributes

If you specify a nonexistent object attribute in the `TRANSLATE` clause, then JPublisher ignores it without issuing a warning message.

Consider the following example from an `INPUT` file:

```
type PERSON translate X as attr1
```

A situation in which `X` is not an attribute of `PERSON` does not cause JPublisher to issue a warning message.

JPublisher Reserved Terms

Do not use any of the following reserved terms as SQL or Java identifiers in the `INPUT` file.

```
AS
GENERATE
IMPLEMENTS
SQLSTATEMENTS_TYPE
SQLSTATEMENTS_METHOD
SQL
TRANSLATE
```

```
TOPLEVEL
TYPE
VERSION
```

A

Generated Code Examples

This appendix contains generated code examples that are explained in the following sections:

- Generated Code: User Subclass for Java-to-Java Transformations
- Generated Code: SQL Statement
- Generated Code: Server-Side Java Call-in

Generated Code: User Subclass for Java-to-Java Transformations

This section contains generated code for the example in "JPublisher-Generated Subclasses for Java-to-Java Type Transformations" on page 4-16. This example uses style files and holder classes in generating a user subclass that supports PL/SQL output arguments and uses Java types supported by Web services. This example shows the JPublisher-generated interface, base class, and user subclass to publish the following PL/SQL package, `foo_pack`, consisting of the stored function `foo`, by using Java types suitable for Web services:

```
CREATE OR REPLACE PACKAGE foo_pack AS
   FUNCTION foo(a IN OUT SYS.XMLTYPE, b INTEGER) RETURN CLOB;
END;
/
```

Assume that you translate the `foo_pack` package as follows:

```
% jpub -u scott/tiger -s foo_pack:FooPack -style=webservices10
```

Note that:

- The SQL type, `xmltype`, is initially mapped to the Java type, `oracle.sql.SimpleXMLType`, in the JPublisher type map.
- `SimpleXMLType` is mapped to `javax.xml.transform.Source` in the `webservices10.properties` style file for use in Web services.

  See Also: "Support for XMLTYPE" on page 3-12

- The holder class for `Source` data, `javax.xml.rpc.holders.SourceHolder`, is used for the output `Source` argument.

  See Also: "Passing Output Parameters in JAX-RPC Holders" on page 4-4

- The style file specifies the generated code naming pattern, "`%2Base:%2User#%2`", related to the following JPublisher command:

  ```
  -s foo_pack:FooPack
  ```

  This results in the generation of interface code in `FooPack.java`, base class code in `FooPackBase.java`, and user subclass code in `FooPackUser.java`.

The `foo()` wrapper method in the `FooPackUser` user subclass uses the following type transformation functionality and a call to the corresponding `_foo()` method of the generated base class, which is where the Java Database Connectivity (JDBC) calls occur to invoke the wrapped stored function, `foo`:

```
foo (SourceHolder, Integer)
{
     SourceHolder -> Source
          Source -> SimpleXMLType
               _foo (SimpleXMLType[], Integer);
          SimpleXMLType -> Source
     Source -> SourceHolder
}
```

Interface Code

The following code is for the Java interface that JPublisher generates in `FooPack.java`:

```
import java.sql.SQLException;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;
// Ensure that the java.io.* package and other required packages are imported.
import java.io.*;

public interface FooPack extends java.rmi.Remote
{
   public java.lang.String foo(SourceHolder _xa_inout_x, Integer b) throws
java.rmi.RemoteException;
}
```

Base Class Code

The following code is for the base class that JPublisher generates in `FooPackBase.java`. The `_foo()` method is called by the `foo()` method of the user subclass and uses JDBC to invoke the `foo` stored function of the `foo_pack` PL/SQL package that JPublisher is publishing.

> **Note:** Comments indicate corresponding SQLJ code, which JPublisher translates automatically during the generation of the class.

```
import java.sql.SQLException;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;
// Ensure that the java.io.* package and other required packages are imported.
import java.io.*;
```

```
public class FooPackBase
{

  /* connection management */
  protected DefaultContext __tx = null;
  protected Connection __onn = null;
  public void _setConnectionContext(DefaultContext ctx) throws SQLException
  { release(); __tx = ctx;
    ctx.setStmtCacheSize(0);
    ctx.setDefaultStmtCacheSize(0);
    if (ctx.getConnection() instanceof oracle.jdbc.OracleConnection)
    {
       try
       {
          java.lang.reflect.Method m =
             ctx.getConnection().getClass().getMethod("setExplicitCachingEnabled",
             new Class[]{Boolean.TYPE});
          m.invoke(ctx.getConnection(), new Object[]{Boolean.FALSE});
       }
       catch(Exception e) { /* do nothing for pre-9.2 JDBC drivers*/ }
    }}
public DefaultContext _getConnectionContext() throws SQLException
{ if (__tx==null)
  { __tx = (__onn==null) ? DefaultContext.getDefaultContext() :
                               new DefaultContext(__onn); }
  return __tx;
};
public Connection _getConnection() throws SQLException
{ return (__onn==null) ? ((__tx==null) ? null : __tx.getConnection()) : __onn; }
public void release() throws SQLException
{ if (__tx!=null && __onn!=null) __tx.close(ConnectionContext.KEEP_CONNECTION);
  __onn = null; __tx = null;
}

/* constructors */
public FooPackBase() throws SQLException
{ __tx = DefaultContext.getDefaultContext();
}
public FooPackBase(DefaultContext c) throws SQLException
{ __tx = c; }
public FooPackBase(Connection c) throws SQLException
{__onn = c; __tx = new DefaultContext(c);  }

/* *** _foo() USES JDBC TO INVOKE WRAPPED foo STORED PROCEDURE *** */

public oracle.sql.CLOB _foo (
  oracle.sql.SimpleXMLType a[],
  Integer b)
  throws SQLException
{
  oracle.sql.CLOB __jPt_result;

  //  ************************************************************
  //  #sql [_getConnectionContext()] __jPt_result = { VALUES(SCOTT.FOO_PACK.FOO(
  //        :a[0],
  //        :b))  };
  //  ************************************************************

{
  // declare temps
```

```
  oracle.jdbc.OracleCallableStatement __sJT_st = null;
  sqlj.runtime.ref.DefaultContext __sJT_cc = _getConnectionContext();
  if (__sJT_cc==null) sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();
  sqlj.runtime.ExecutionContext.OracleContext __sJT_ec =
              ((__sJT_cc.getExecutionContext()==null) ?
              sqlj.runtime.ExecutionContext.raiseNullExecCtx() :
              __sJT_cc.getExecutionContext().getOracleContext());
  try {
   String theSqlTS = "BEGIN :1 := SCOTT.FOO_PACK.FOO
              (\n       :2 ,\n       :3 )  \n; END;";
   __sJT_st = __sJT_ec.prepareOracleCall(__sJT_cc,"0FooPackBase",theSqlTS);
   if (__sJT_ec.isNew())
   {
      __sJT_st.registerOutParameter(1,oracle.jdbc.OracleTypes.CLOB);
      __sJT_st.registerOutParameter(2,2007,"SYS.XMLTYPE");
   }
   // set IN parameters
   if (a[0]==null) __sJT_st.setNull(2,2007,"SYS.XMLTYPE");
   else __sJT_st.setORAData(2,a[0]);
   if (b == null) __sJT_st.setNull(3,oracle.jdbc.OracleTypes.INTEGER);
   else __sJT_st.setInt(3,b.intValue());
 // execute statement
   __sJT_ec.oracleExecuteUpdate();
   // retrieve OUT parameters
   __jPt_result =  (oracle.sql.CLOB) __sJT_st.getCLOB(1);
   a[0] = (oracle.sql.SimpleXMLType)__sJT_st.getORAData
                                (2,oracle.sql.SimpleXMLType.getORADataFactory());
  } finally { __sJT_ec.oracleClose(); }
}

 //  *******************************************************

    return __jPt_result;
  }
}
```

User Subclass Code

The following code is for the user subclass that JPublisher generates in `FooPackUser.java`. This class extends the `FooPackBase` class and implements the `FooPack` interface. The `foo()` method calls the `_foo()` method of the base class. Java-to-Java transformations are handled in `try` blocks, as indicated in code comments.

```
import java.sql.SQLException;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;
// Ensure that the java.io.* package and other required packages are imported.
import java.io.*;

public class FooPackUser extends FooPackBase implements FooPack, java.rmi.Remote
{

  /* constructors */
  public FooPackUser() throws SQLException  { super(); }
  public FooPackUser(DefaultContext c) throws SQLException { super(c); }
  public FooPackUser(Connection c) throws SQLException { super(c); }
  /* superclass methods */
```

```
  public java.lang.String foo(SourceHolder _xa_inout_x, Integer b)
                                                    throws java.rmi.RemoteException
  {
    oracle.sql.CLOB __jRt_0 = null;
    java.lang.String __jRt_1 = null;

/* *** FOLLOWING try BLOCK CONVERTS SourceHolder TO Source *** */

    try {
       javax.xml.transform.Source[] a_inout;
       // allocate an array for holding the OUT value
       a_inout = new javax.xml.transform.Source[1];
       if (_xa_inout_x!=null) a_inout[0] = _xa_inout_x.value;
       oracle.sql.SimpleXMLType[] xa_inoutx;
       xa_inoutx = new oracle.sql.SimpleXMLType[1];

/* *** FOLLOWING try BLOCK TRANSFORMS Source TO SimpleXMLType *** */

    try
    {
      javax.xml.transform.Transformer trans =
           javax.xml.transform.TransformerFactory.newInstance().newTransformer();
      xa_inoutx[0] = null;
      if (a_inout[0]!=null)
      {
        java.io.ByteArrayOutputStream buf = new java.io.ByteArrayOutputStream();
        javax.xml.transform.stream.StreamResult streamr =
                                new javax.xml.transform.stream.StreamResult(buf);
        trans.transform(a_inout[0], streamr);
        xa_inoutx[0] = new oracle.sql.SimpleXMLType(_getConnection());
        xa_inoutx[0] = xa_inoutx[0].createxml(buf.toString());
      }
    }
    catch (java.lang.Throwable t)
    {
      throw OC4JWsDebugPrint(t);
    }

/* *** CALL _foo() FROM BASE CLASS (SUPER CLASS) *** */

    __jRt_0 = super._foo(xa_inoutx, b);

/* *** FOLLOWING try BLOCK TRANSFORMS SimpleXMLType TO Source *** */

    try
    {
      javax.xml.parsers.DocumentBuilder db =
      javax.xml.parsers.DocumentBuilderFactory.newInstance().newDocumentBuilder();
      a_inout[0] = null;
      if (xa_inoutx[0]!=null)
      {
        org.w3c.dom.Document _tmpDocument_ = db.parse
       (new java.io.ByteArrayInputStream(xa_inoutx[0].getstringval().getBytes()));
        a_inout[0]= new javax.xml.transform.dom.DOMSource(_tmpDocument_);
      }
    }
    catch (java.lang.Throwable t)
    {
      throw OC4JWsDebugPrint(t);
    }
```

```
/* *** FOLLOWING CODE CONVERTS Source TO SourceHolder *** */

      // convert OUT value to a holder
      if (a_inout!=null) _xa_inout_x.value = a_inout[0];
    if (__jRt_0==null)
    {
       __jRt_1=null;
    }
    else
    {
       __jRt_1=readerToString(__jRt_0.getCharacterStream());
    }

    }
    catch (Exception except) {
        try {
           Class sutil = Class.forName("com.evermind.util.SystemUtils");
           java.lang.reflect.Method getProp = sutil.getMethod("getSystemBoolean",
                                      new Class[]{String.class, Boolean.TYPE});
           if (((Boolean)getProp.invoke(null, new Object[]{"ws.debug",
                       Boolean.FALSE})).booleanValue()) except.printStackTrace();
        }  catch (Throwable except2) {}
        throw new java.rmi.RemoteException(except.getMessage(), except);
    }
    return __jRt_1;
  }
 private java.lang.String readerToString(java.io.Reader r)
                                                 throws java.sql.SQLException
{
        CharArrayWriter caw = new CharArrayWriter();

        try
          {
                //Read from reader and write to writer
                boolean done = false;

                while (!done)
                {
                        char[] buf = new char[4096];
                        int len = r.read(buf, 0, 4096);
                        if(len == -1)
                        {
                        done = true;
                        }
                        else
                        {
                                caw.write(buf,0,len);
                        }
                }
          }
         catch(Throwable t)
         {
           throw OC4JWsDebugPrint(t);
         }
        return caw.toString();
 }
    private void populateClob(oracle.sql.CLOB clb, java.lang.String data)
                                                          throws Exception
    {
```

```
        java.io.Writer writer = clb.getCharacterOutputStream();
        writer.write(data.toCharArray());
        writer.flush();
        writer.close();
    }
    private boolean OC4JWsDebug()
    {
      boolean debug = false;
      try {
        // Server-side Debug Info for "java -Dws.debug=true -jar oc4j.jar"
        Class sutil = Class.forName("com.evermind.util.SystemUtils");
        java.lang.reflect.Method getProp = sutil.getMethod("getSystemBoolean",
                                      new Class[]{String.class, Boolean.TYPE});
        if (((Boolean)getProp.invoke(null, new Object[]{"ws.debug",
                                  Boolean.FALSE})).booleanValue())
        {
          debug = true;
        }
      }  catch (Throwable except2) {}
      return debug;
    }
    private java.sql.SQLException OC4JWsDebugPrint(Throwable t)
    {
      java.sql.SQLException t0 =  new java.sql.SQLException(t.getMessage());
      if (!OC4JWsDebug()) return t0;
      t.printStackTrace();
      try
      {
        java.lang.reflect.Method getST =
                      Exception.class.getMethod("getStackTrace", new Class[]{});
        java.lang.reflect.Method setST =
                      Exception.class.getMethod("setStackTrace", new Class[]{});
        setST.invoke(t0, new Object[]{getST.invoke(t, new Object[]{})});
      }
      catch (Throwable th){}
      return t0;
    }
}
```

Generated Code: SQL Statement

This section contains a generated code example for a specified SQL statement, related to the discussion in "Declaration of SQL Statements to Translate" on page 6-17.

The example is for the following sample settings of the `-sqlstatement` option:

```
-sqlstatement.class=MySqlStatements
-sqlstatement.getEmp="select ename from emp
                      where ename=:{myname VARCHAR}"
-sqlstatement.return=both
```

Note that for this example:

- Code comments show `#sql` statements that correspond to the translated code shown.
- The `getEmpBeans()` method, generated because of the `-sqlstatement.return=both` setting, returns an array of JavaBeans. Each element represents a row of the result set. The `GetEmpRow` class is defined for this purpose.

- JPublisher generates a SQLJ class. The result set is mapped to a SQLJ iterator.

(For `UPDATE`, `INSERT`, or `DELETE` statements, code is generated both with and without batching for array binds.)

The translated SQLJ code that JPublisher would produce is:

```
public class MySqlStatements_getEmpRow
{

  /* connection management */

  /* constructors */
  public MySqlStatements_getEmpRow()
  { }

  public String getEname() throws java.sql.SQLException
  { return ename; }

  public void setEname(String ename) throws java.sql.SQLException
  { this.ename = ename; }

  private String ename;
}

/*@lineinfo:filename=MySqlStatements*/
/*@lineinfo:user-code*/
/*@lineinfo:1^1*/
import java.sql.SQLException;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;
import oracle.sql.*;

public class MySqlStatements
{

  /* connection management */
  protected DefaultContext __tx = null;
  protected Connection __onn = null;
  public void setConnectionContext(DefaultContext ctx) throws SQLException
  { release(); __tx = ctx; }
  public DefaultContext getConnectionContext() throws SQLException
  { if (__tx==null)
    { __tx = (__onn==null) ? DefaultContext.getDefaultContext() :
                             new DefaultContext(__onn); }
    return __tx;
  };
  public Connection getConnection() throws SQLException
  { return (__onn==null) ? ((__tx==null) ? null : __tx.getConnection()) : __onn; }
  public void release() throws SQLException
  { if (__tx!=null && __onn!=null) __tx.close(ConnectionContext.KEEP_CONNECTION);
    __onn = null; __tx = null;
  }

  /* constructors */
  public MySqlStatements() throws SQLException
  { __tx = DefaultContext.getDefaultContext(); }
  public MySqlStatements(DefaultContext c) throws SQLException
  { __tx = c; }
  public MySqlStatements(Connection c) throws SQLException
```

```
  {__onn = c; __tx = new DefaultContext(c); }
/*@lineinfo:generated-code*/
/*@lineinfo:36^1*/

//  ************************************************************
//  SQLJ iterator declaration:
//  ************************************************************

public static class getEmpIterator
    extends sqlj.runtime.ref.ResultSetIterImpl
    implements sqlj.runtime.NamedIterator
{
  public getEmpIterator(sqlj.runtime.profile.RTResultSet resultSet)
    throws java.sql.SQLException
  {
    super(resultSet);
    enameNdx = findColumn("ename");
    m_rs = (oracle.jdbc.OracleResultSet) resultSet.getJDBCResultSet();
  }
  private oracle.jdbc.OracleResultSet m_rs;
  public String ename()
    throws java.sql.SQLException
  {
    return m_rs.getString(enameNdx);
  }
  private int enameNdx;
}

//  ************************************************************

/*@lineinfo:user-code*/
/*@lineinfo:36^56*/

  public MySqlStatements_getEmpRow[] getEmpBeans (String myname)
         throws SQLException
  {
    getEmpIterator iter;
    /*@lineinfo:generated-code*/
    /*@lineinfo:43^5*/
//  ************************************************************
//  #sql [getConnectionContext()]
//                      iter = { select ename from emp where ename=:myname };
//  ************************************************************
{
  // declare temps
  oracle.jdbc.OraclePreparedStatement __sJT_st = null;
  sqlj.runtime.ref.DefaultContext __sJT_cc = getConnectionContext();
  if (__sJT_c c==null) sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();
  sqlj.runtime.ExecutionContext.OracleContext __sJT_ec =
        ((__sJT_cc.getExecutionContext()==null) ?
        sqlj.runtime.ExecutionContext.raiseNullExecCtx() :
        __sJT_cc.getExecutionContext().getOracleContext());
  try {
   String theSqlTS = "select ename from emp where ename= :1";
   __sJT_st = __sJT_ec.prepareOracleStatement
                    (__sJT_cc,"0MySqlStatements",theSqlTS);
   // set IN parameters
   __sJT_st.setString(1,myname);
   // execute query
   iter = new MySqlStatements.getEmpIterator
```

```
            (new sqlj.runtime.ref.OraRTResultSet
            (__sJT_ec.oracleExecuteQuery(),__sJT_st,"0MySqlStatements",null));
  } finally { __sJT_ec.oracleCloseQuery(); }
}

//  ************************************************************

/*@lineinfo:user-code*/
/*@lineinfo:43^84*/
    java.util.Vector v = new java.util.Vector();
    while (iter.next())
    {
       MySqlStatements_getEmpRow r = new MySqlStatements_getEmpRow();
       r.setEname(iter.ename());
       v.addElement(r);
    }
    MySqlStatements_getEmpRow[] __jPt_result =
          new MySqlStatements_getEmpRow[v.size()];
    for (int i = 0; i < v.size(); i++)
       __jPt_result[i] = (MySqlStatements_getEmpRow) v.elementAt(i);
    return __jPt_result;
  }

  public java.sql.ResultSet getEmp (String myname)
          throws SQLException
  {
    sqlj.runtime.ResultSetIterator iter;
    /*@lineinfo:generated-code*/
    /*@lineinfo:62^5*/

//  ************************************************************
//  #sql [getConnectionContext()] iter =
//                      { select ename from emp where ename=:myname };
//  ************************************************************

{
  // declare temps
  oracle.jdbc.OraclePreparedStatement __sJT_st = null;
  sqlj.runtime.ref.DefaultContext __sJT_cc = getConnectionContext();
  if (__sJT_c c==null) sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();
  sqlj.runtime.ExecutionContext.OracleContext __sJT_ec =
             ((__sJT_cc.getExecutionContext()==null) ?
             sqlj.runtime.ExecutionContext.raiseNullExecCtx() :
             __sJT_cc.getExecutionContext().getOracleContext());
  try {
   String theSqlTS = "select ename from emp where ename= :1";
   __sJT_st = __sJT_ec.prepareOracleStatement
                (__sJT_cc,"1MySqlStatements",theSqlTS);
   // set IN parameters
   __sJT_st.setString(1,myname);
   // execute query
   iter = new sqlj.runtime.ref.ResultSetIterImpl
         (new sqlj.runtime.ref.OraRTResultSet
         (__sJT_ec.oracleExecuteQuery(),__sJT_st,"1MySqlStatements",null));
  } finally { __sJT_ec.oracleCloseQuery(); }
}

//  ************************************************************

/*@lineinfo:user-code*/
```

```
/*@lineinfo:62^84*/
    java.sql.ResultSet __jPt_result = iter.getResultSet();
    return __jPt_result;
  }
}
/*@lineinfo:generated-code*/
```

Generated Code: Server-Side Java Call-in

JPublisher supports the calling of Java methods in the database from a Java client outside the database. In Oracle Database 10*g* release 2 (10.2), the JPublisher `-dbjava` option is used for server-side Java invocation. Unlike the `-java` option, the `-dbjava` option supports non-serializable parameter or return types.

See Also: "Server-Side Java Invocation (Call-in)" on page 5-3

This section describes an example of server-side Java invocation. This section comprises:

- The Source Files
- Publishing Server-Side Java Class
- The Generated Files
- Testing the Published Files

Note: You must have the 10*g* Release 1 (10.2) version of the Oracle Database.

The Source Files

In this example, there are three source files to be created. These are:

- A server-side Java class
- A JavaBean used in the server-side Java class
- An entry point Java class that invokes the methods in the published classes

The source code of these files is as follows:

Server-Side Java Class

The source code of the server-side Java class, `Callin2.java`, is as follows:

```
public class Callin2
{
  public static int testInt(int i)
  { return i; }
  public static int[] testInt(int[] i)
  { return i; }
  public static int[][] testInt(int[][] i)
  { return i; }
  public static Integer testInteger(Integer i)
  { return i; }
  public static Integer[] testInteger(Integer[] i)
  { return i; }
  public static Integer[][] testInteger(Integer[][] i)
  { return i; }
```

```
  // Test ORAData
  public static oracle.sql.NUMBER testNum(oracle.sql.NUMBER num)
  { return num; }
  public oracle.sql.NUMBER testInstNum(oracle.sql.NUMBER num)
  { return num; }
  public oracle.sql.NUMBER[] testInstNum(oracle.sql.NUMBER[] num)
  { return num; }
  public oracle.sql.NUMBER[][] testInstNum(oracle.sql.NUMBER[][] num)
  { return num; }

  // Test Beans
  public static Callin2Bean testBean()
  { return new Callin2Bean("mybean", new int[]{1,2}); }
  public static Callin2Bean testBean (Callin2Bean b)
  { return b; }
  public static Callin2Bean[] testBean (Callin2Bean[] b)
  { return b; }
  public static Callin2Bean[][] testBean (Callin2Bean[][] b)
  { return b; }
  public Callin2Bean testInstBean (Callin2Bean b)
  { return b; }
  public Callin2Bean[] testInstBean (Callin2Bean[] b)
  { return b; }
  public Callin2Bean[][] testInstBean (Callin2Bean[][] b)
  { return b; }

  // Test Serializable
  public static java.io.Serializable testSer()
  { return new String("test Serializable"); }
  public static java.io.Serializable testSer (java.io.Serializable b)
  { return b; }
  public static java.io.Serializable[] testSer (java.io.Serializable[] b)
  { return b; }
  public static java.io.Serializable[][] testSer (java.io.Serializable[][] b)
  { return b; }
  public java.io.Serializable testInstSer (java.io.Serializable b)
  { return b; }
  public java.io.Serializable[] testInstSer (java.io.Serializable[] b)
  { return b; }
  public java.io.Serializable[][] testInstSer (java.io.Serializable[][] b)
  { return b; }
}
```

JavaBean Used in the Server-Side Java Class

The source code of the JavaBean, `Callin2Bean.java`, used in the server-side Java class, `Callin2.java`, is as follows:

```
public class Callin2Bean
{
  private String stringValue = "";
  private int[] numberValue;

  public Callin2Bean ()
  {
  }

  public Callin2Bean(String string_val, int[] number_val)
  {
    stringValue = string_val;
```

```
    numberValue = number_val;
  }

  public void setStringValue(String string_val)
  {
    stringValue = string_val;
  }

  public String getStringValue ()
  {
    return stringValue;
  }

  public void setNumberValue (int[] number_val)
  {
    numberValue = number_val;
  }

  public int[] getNumberValue ()
  {
    return numberValue;
  }

  public boolean equals(Object other)
  {
    if(other instanceof Callin2Bean)
    {
      Callin2Bean my_bean = (Callin2Bean)other;
      if ( stringValue.equals(my_bean.getStringValue()) &&
compareIntArray(numberValue, my_bean.getNumberValue()) )
      {
        return true;
      }
    }
    return false;
  }

  private boolean compareIntArray(int[] b1, int[] b2)
  {
    try
    {
      if ((b1 == null) && (b2 == null))
        return true;

      if ((b1.length == 0) && (b2.length == 0))
        return true;

      if (b1.length != b2.length)
        return false;
      int x;
      for (x = 0; x < b1.length; x++)
      {
        if (b1[x] != b2[x])
          return false;
      }
      return true;
    }
    catch (Exception e)
    {
      return false;
```

```
    }
  }
}
```

Entry Point Java Class

The `TestCallin2.java` is the entry point Java class for this example. This class invokes the methods in the published class. The source code of the `TestCallin2.java` file is as follows:

```
public class TestCallin2
{
  public static void main(String[] args) throws Exception
  {
    java.sql.DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
    oracle.jdbc.OracleConnection conn = (oracle.jdbc.OracleConnection)

//java.sql.DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:lsqlj1",
"scott", "tiger");
    java.sql.DriverManager.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");

   Callin2Client tkpu = new Callin2Client (conn);

   System.out.println("testInstNum() returned " + tkpu.testinstnum(new
java.math.BigDecimal(1999)));

   TblNumber na = new TblNumber(new java.math.BigDecimal[]{new
java.math.BigDecimal(2999)});
   System.out.println("testInstNum([]) returned " +
   tkpu.testinstnum(na).getArray()[0]);

   ObjCallin2bean mb = new ObjCallin2bean("mybean", na);
   System.out.println("testCallin2Bean() returned " +
tkpu.testbean(mb).getStringvalue());
   System.out.println("testCallin2Bean([]) returned " + tkpu.testbean(new
TblObjCallin2bean(new ObjCallin2bean[]{mb})).getArray()[0].getStringvalue());

   java.io.Serializable s = new java.util.Hashtable();
   ((java.util.Hashtable) s).put("bush", "cheny");
   ((java.util.Hashtable) s).put("kerry", "dean");
   java.io.Serializable[] s1 = new java.io.Serializable[]{s};
   java.io.Serializable[][] s2 = new java.io.Serializable[][]{s1};
   System.out.println("testSer() returned " + ((java.util.Hashtable)
tkpu.testser(s)).get("kerry"));
   System.out.println("testSer([]) returned " + ((java.util.Hashtable)
tkpu.testser0(s1)[0]).get("kerry"));
   System.out.println("testSer([][]) returned " + ((java.util.Hashtable)
tkpu.testser1(s2)[0][0]).get("kerry"));
  }
}
```

Note: If you are connecting to the database using the JDBC Thin driver, then you need to uncomment the first call to the `getConnection()` method in the preceding code and comment the second call to the `getConnection()` method, which includes a connect statement for an Oracle Call Interface (OCI) driver.

Publishing Server-Side Java Class

After you have created the required source files, you need to publish these files. To publish the server-side Java classes, you need to first load these files on to the database. Ensure that you load both `Callin2.java` and `Callin2Bean.java`. The command for loading the files is:

```
% loadjava -u scott/tiger -r -v -f Callin2.java Callin2Bean.java
```

To publish these files, issue the following command:

```
% jpub -u scott/tiger -sysuser=sys/change_on_install -dbjava=Callin2:Callin2Client
-dir=tmp
```

The JPublisher output is:

```
tmp/Callin2JPub.java
tmp/plsql_wrapper.sql
tmp/plsql_dropper.sql
SCOTT.TBL_NUMBER
SCOTT.TBL_TBL_NUMBER
SCOTT.OBJ_CALLIN2BEAN
SCOTT.TBL_OBJ_CALLIN2BEAN
SCOTT.TBL_TBL_OBJ_CALLI
SCOTT.JPUB_PLSQL_WRAPPER
Executing tmp/plsql_dropper.sql
Executing tmp/plsql_wrapper.sql
Loading Callin2JPub.java
```

The Generated Files

When you publish the server-side Java classes JPublisher generates a few Java classes and PL/SQL scripts. Some of these files are:

- `Callin2JPub.java`

  This is the server-side Java wrapper class for `Callin2.java`.

- `Callin2Client.java`

  This is the client-side Java class.

- `plsql_wrapper.sql`

  This is the PL/SQL wrapper for the server-side Java class.

- `plsql_dropper.sql`

  This is the PL/SQL script dropping the PL/SQL wrapper.

The other files generated by JPublisher are as follows:

- `TblNumber.java`

  A client-side wrapper generated for `int[]`.

- `TblTblNumber.java`

  A client-side wrapper generated for `int[][]`.

- `ObjCallin2bean.java`

  A client-side wrapper generated for the server-side class `Callin2Bean`.

- `ObjCallin2beanRef.java`

A client-side wrapper generated for the server-side class `Callin2Bean`, used as a `REF` column in a table. This file is generated, but not used in the call-in scenario.

- `TblObjCallin2bean.java`

  A client-side wrapper generated for `Callin2Bean[]`.

- `TblTblObjCalli.java`

  A client-side wrapper generated for `Callin2Bean[][]`.

Testing the Published Files

After the files have been published, you can test the published classes by issuing the following commands:

```
% javac -classpath tmp:${CLASSPATH} -d tmp TestCallin2.java
% java -classpath tmp:${CLASSPATH} TestCallin2
```

B

Troubleshooting

This chapter covers the troubleshooting tips for JPublisher. It contains the following sections:

Error While Publishing Web Services Client

When publishing Web services client using the `-proxywsdl` opting, you may come across one of the following errors:

```
java.lang.Exception: Error compiling generated client proxy
Can't load library "jdk12/jre/lib/i386/libjava.so", because
jdk12/jre/lib/i386/libjava.so: symbol __libc_wait, version
GLIBC_2.0 not defined in file libc.so.6 with link time reference
Could not create the Java virtual machine.
```

Or

```
% jpub -proxywsdl=...

Method getStackTrace() not found in class java.lang.Exception.
         _e.setStackTrace(e.getStackTrace());
                                            ^
1 error
```

This problem is caused by Java Development Kit (JDK) 1.2 used in the environment. You need to include JDK 1.4 in the `PATH` environment variable to resolve this error.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X