Oracle® Database

JPublisher User's Guide

10*g* Release 1 (10.1)

Part No. B10983-01

December 2003

ORACLE®

Oracle Database JPublisher User's Guide, 10*g* Release 1 (10.1)

Part No. B10983-01

Primary Author: Brian Wright

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

Contributors: Quan Wang, Prabha Krishna, Ellen Siegal

Contents

Send Us Your Comments

Oracle Database JPublisher User's Guide, 10*g* Release 1 (10.1)

Part No. B10983-01

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

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

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

- Electronic mail: jpgreader_us@oracle.com
- FAX: (650) 506-7225. Attn: Java Platform Group, Information Development Manager
- Postal service:

  Oracle Corporation
  Java Platform Group, Information Development Manager
  500 Oracle Parkway, Mailstop 4op9
  Redwood Shores, CA 94065
  USA

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

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

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 contains these topics:

Intended Audience

The *Oracle Database JPublisher User's Guide* is intended for JDBC and 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 DML statements as methods in Java classes
- Create Java and PL/SQL wrappers for Web services client proxy classes, to enable call-outs to Web services from the database
- Publish server-side SQL, PL/SQL or Java entities as Web services, to enable call-ins from outside the database

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. 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 additional information, visit the Oracle Accessibility Program Web site at

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

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

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

Structure

This document contains:

Chapter 1, "Introduction to JPublisher"

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

Chapter 2, "Datatype and Java-to-Java Type Mappings"

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

Chapter 3, "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 to use the generated classes and interfaces.

Chapter 4, "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 5, "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.

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, thereby improving 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 (Java Database Connectivity). This includes an overview of the Oracle JDBC drivers, details of the Oracle implementation of JDBC 1.22, 2.0, and 3.0 features, and 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 User'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 XML 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*
- *PL/SQL Packages and Types Reference*
- *PL/SQL User's Guide and Reference*

- *Oracle Database SQL Reference*
- *Oracle Net Services Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*
- *Oracle Database Globalization Support Guide*
- *Oracle Database Reference*

Note: Oracle error message documentation is available in HTML only. If you have access to the Oracle Documentation CD only, you can browse the error messages by range. Once 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 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 more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.	*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, datatypes (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.
`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`*

`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 introduction and overview of the JPublisher utility, concluding with a summary of JPublisher operations and a sample translation. It covers the following topics:

Overview of JPublisher

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

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

- User-defined SQL object types
- Object reference types (REF types)
- User-defined SQL collection types (VARRAY types or nested table types)
- PL/SQL packages
- Server-side Java classes
- SQL queries and 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* is used where a particular Java type is associated with a given user-defined SQL type such as an object type (for example, a Person class for a corresponding PERSON SQL type). Additionally, there is a particular corresponding Java type for each attribute of the SQL object type.

The utility generates `get`*`XXX`*`()` and `set`*`XXX`*`()` accessor methods for each attribute of an object type. If your object types have 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 that executes in Oracle Database.

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

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

- Extend the generated classes. This process is straightforward, because JPublisher can also generate initial versions of the subclasses for you, to which you can add your desired functionality.
- Write your own Java classes by hand, without using JPublisher. 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. Typically, you would use this approach if you must be able to generically process *any* SQL object, collection, reference, or OPAQUE type.

In addition, JPublisher simplifies access to PL/SQL types from Java. You can employ predefined or user-defined mappings between PL/SQL and SQL types, as well as make use of PL/SQL conversion functions between such types. With these mappings in place, JPublisher can automatically generate all the required Java and PL/SQL code.

Paralleling the functionality of publishing SQL or PL/SQL entities to Java, it is also possible to publish server-side Java to client-side Java, effectively allowing your application to make direct calls to Java classes in the database.

Several features enable exposure of generated Java classes—from publishing either SQL or PL/SQL entities or server-side Java entities—as Web services. See "Summary of JPublisher Support for Web Services" on page 4-1 for an overview of these features.

JPublisher Initial Considerations

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

- New JPublisher Features in Oracle Database 10g
- JPublisher Usage of the Oracle SQLJ Implementation
- JPublisher General Requirements
- Required Packages and JAR Files in the Database
- JPublisher Limitations

New JPublisher Features in Oracle Database 10*g*

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

- New Features for Web Services
- Awareness of Java Environment Classpath
- New Features for Usage of SQLJ

New Features for Web Services

JPublisher is used in publishing from SQL or PL/SQL to Java, or in publishing from server-side Java to client-side Java, with new features in place to enable exposure of generated Java classes as Web services for invocation from outside the database. There are two stages to this:

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

There are also new features to load and wrap client proxy classes so that external Web services can be invoked from Java or PL/SQL inside the database.

New JPublisher features in Oracle Database 10*g* include the following. For a more detailed overview, including how these features relate to Web services, see "Summary of JPublisher Support for Web Services" on page 4-1.

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

Also see "Publishing Server-Side Java Classes" on page 1-16 and "Publishing SQL Queries or DML Statements" on page 1-16.

Awareness of Java Environment Classpath

Prior to Oracle Database 10*g*, UNIX releases of JPublisher ignored the environment classpath, instead using a classpath provided through the JPublisher command line that included the required JPublisher and JDBC classes. In Oracle Database 10*g*, the environment classpath is appended to the classpath that is provided through the command line.

On all platforms now, JPublisher picks up the environment classpath. This feature ensures successful execution in circumstances in which JPublisher must be able to load user-provided types, such as for Web services call-outs and the 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. For more information, see "Options to Facilitate Web Services Call-Outs" on page 5-34 and "Code Generation for Wrapper Class and PL/SQL Wrapper Options" on page 5-46. (By 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.)

Awareness of the environment classpath also plays a role in how JPublisher can represent query results. See "Mapping of REF CURSOR Types and Result Sets" on page 2-7.

New Features for Usage of SQLJ

In most cases, such as whenever wrapper methods are required, JPublisher generates code that uses the Oracle SQLJ implementation. But in Oracle Database 10*g*, the use of

SQLJ is now transparent to the user by default. The next section, "JPublisher Usage of the Oracle SQLJ Implementation", describes this.

JPublisher Usage of the Oracle SQLJ Implementation

The following sections describe when and how JPublisher uses SQLJ, provide an overview of SQLJ, and discuss backward-compatibility modes that relate to the generation of SQLJ source files:

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

Note: The Oracle SQLJ translator and runtime libraries are supplied with the JPublisher product.

Overview of SQLJ Usage

The JPublisher utility uses the Oracle SQLJ ("SQL in Java") implementation, generating SQLJ code as an intermediate step in most circumstances—whenever wrapper methods are created, either for classes representing PL/SQL packages or for classes representing SQL object types that define methods (PL/SQL stored procedures). In these circumstances, JPublisher uses the Oracle SQLJ translator during compilation, and the Oracle SQLJ runtime during program execution.

In Oracle Database 10*g*, as a convenience, JPublisher usage of SQLJ is transparent by default. SQLJ source files that JPublisher generates are automatically translated and deleted unless you specify JPublisher settings to choose otherwise. This automatic translation saves you the step of explicitly translating the files. The resulting `.java` files that use SQLJ functionality, and the associated `.class` files produced by compilation, define what are still referred to as *SQLJ classes*. These classes use the Oracle SQLJ runtime APIs during execution. Generated classes that do not use the SQLJ runtime 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.

For those familiar with SQLJ command-line options, it is possible in Oracle Database 10*g* to pass options to the SQLJ translator through the JPublisher `-sqlj` option. See "Option to Access SQLJ Functionality" on page 5-42.

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 runtime library for JDK 1.2 and higher.

Overview of SQLJ Concepts

A SQLJ program is a Java program containing embedded SQL statements that comply with the 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. The use of SQLJ was chosen because of the simplified code that SQLJ uses for database access, compared to JDBC code. In SQLJ, a SQL statement is embedded in a single `#sql` statement, while several JDBC statements may be required for the same operation.

Because JPublisher generates code that uses SQLJ features, this document discusses some SQLJ concepts. This section briefly defines some key concepts, for those not already familiar with 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, meaning that all the connections you define using a particular connection context class will use tables, views, and stored procedures that share names and datatypes 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 this permits 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 connection context class used by default is `sqlj.runtime.ref.DefaultContext`. The SQLJ *default context* is a default connection object and is 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 will instantiate for database connections. See "SQLJ Connection Context Classes (-context)" on page 5-15.

- 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 first and foremost to take query results from a `SELECT` statement. The strong typing is based on the datatype 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 executed. An execution context instance is associated either implicitly or explicitly with each SQL operation that is executed through SQLJ code.

Backward Compatibility Modes Affecting SQLJ Source Files

In Oracle8*i* and Oracle9*i*, JPublisher produced `.sqlj` source files as visible output, which you could then translate yourself using the SQLJ command-line interface (the `sqlj` script in UNIX or the `sqlj.exe` program in Microsoft Windows).

In Oracle Database 10*g*, JPublisher supports several backward-compatibility settings, through its `-compatible` option, that allow you to continue to work with generated `.sqlj` files in similar fashion:

- To have JPublisher skip the step of translating `.sqlj` files, so that you can translate them explicitly, set `-compatible=sqlj`. Then, to translate the files, you can either run JPublisher again using only the `-sqlj` option (as described in "Option to Access SQLJ Functionality" on page 5-42) or you can run the SQLJ translator directly through its own command-line interface.

- To have JPublisher use "Oracle9*i* compatibility mode", set `-compatible=9i`. This setting results in JPublisher generating `.sqlj` files with the same code as in Oracle9*i* versions. Then you can work directly with the `.sqlj` files.

- To have JPublisher use "Oracle8*i* compatibility mode", set `-compatible=both8i` or `-compatible=8i`. This setting results in JPublisher generating `.sqlj` files with the same code as in Oracle8*i* versions. As with 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 `.sqlj` files, then use the `sqlj` setting. For more information, see "Backward Compatibility and Migration" on page 4-4 and "Backward Compatibility Option" on page 5-43.

JPublisher General Requirements

This section describes the base requirements for JPublisher, then 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 Java Developer's Kit (JDK), among other things.

To use all features of JPublisher, you must generally have the following installed and in your classpath, as applicable:

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

  The `jpub` script (for UNIX) or `jpub.exe` program (for Microsoft Windows) must be in your file path. They are typically in *`ORACLE_HOME`*`/bin` (or *`ORACLE_HOME`*`/sqlj/bin` for manual downloads). With proper setup, if you type just "`jpub`" in the command line, you will see information about common JPublisher option and input settings.
- JPublisher and SQLJ translator classes

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

  Notes:

  - The translator library is also automatically loaded into the database, in`translator-jserver.jar`.
  - The client-side translator library includes JPublisher client-side runtime classes, particularly `oracle.jpub.reflect.Client` for Java call-ins to the database.
  - The database translator library includes JPublisher server-side runtime classes, particularly `oracle.jpub.reflect.Server`, also for Java call-ins to the database.

- SQLJ runtime classes

  The SQLJ runtime library is `runtime12.jar`, for JDK 1.2 or higher. It is typically located in *`ORACLE_HOME`*`/sqlj/lib`.
- Oracle Database 10*g* or Oracle9*i* JDBC drivers

  The Oracle JDBC library—`classes12.jar` for JDK 1.2 or higher, or `ojdbc14.jar` for JDK 1.4 specifically—is typically in *`ORACLE_HOME`*`/jdbc/lib`. See the *Oracle Database JDBC Developer's Guide and Reference* for more information about the JDBC files.

  Each JDBC library also includes the JPublisher runtime classes in the `oracle.jpub.runtime` package.
- Web services classes

  These classes are included in the library `utl_dbws.jar`, typically located in *`ORACLE_HOME`*`/sqlj/lib`.
- Additional PL/SQL packages and 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 are preloaded and some must be loaded manually. See the next section, "Required Packages and JAR Files in the Database".

- JDK version 1.2 or higher (JDK 1.4 or higher for Web services call-outs or to map `SYS.XMLType` for Web services)

  Note that you must be able to invoke the Java compiler, `javac`, from the command line. For information about how to specify a JDK version and a compiler version other than the default for the JPublisher environment, see "Java Environment Options" on page 5-45.

Required Packages and JAR Files in the Database

Some or all of the following PL/SQL packages and JAR files must be present in the database, depending on what JPublisher features you use. Subsections that follow discuss how to verify the presence of these packages and files, and how to load them if they are not present.

- `SQLJUTL` package, to support PL/SQL types
- `SQLJUTL2` package, to support invocation of server-side Java classes
- `UTL_DBWS` package, to support Web services call-outs
- `utl_dbws_jserver.jar` file, to support JAX-RPC or SOAP client proxy classes for Web services call-outs from Oracle Database 10*g*

  See "Options to Facilitate Web Services Call-Outs" on page 5-34 for information about related JPublisher features.
- JAR files to support SOAP client proxy classes for Web services call-outs from Oracle9*i* or Oracle8*i* databases

  For Web services call-outs from Oracle9*i* or Oracle8*i*, there is not yet a convenience JAR file to parallel `utl_dbws_jserver.jar`. You must load several JAR files instead. Also note that JPublisher does not yet support JAX-RPC client proxy classes in Oracle9*i* or Oracle8*i*.
- `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 VM, so the `sqljutl.jar` file is unnecessary. In Oracle9*i* or Oracle8*i*, you must load the file manually.

Note: The `UTL_DBWS` package and `utl_dbws_jserver.jar` file are associated with each other, both supporting 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.

Verifying or Installing the UTL_DBWS Package

In Oracle Database 10*g*, the PL/SQL package `UTL_DBWS` 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 response 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 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
```

JPublisher output such as the following indicates that 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 into 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 utl_dbws_jserver.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/utl_dbws_jserver.jar`

It is not preloaded, but you can verify whether it has already been 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 has already been loaded:

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

If it has not already been loaded, you can use the `loadjava` utility, such as in the following example:

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

Note: Before loading this file, verify that the database `java_pool_size` parameter has a setting of at least 96 MB, and that the `shared_pool_size` parameter has a setting of at least 80 MB. If this is not the case, update the database parameter file (such as `init.ora`) to give these two entries appropriate settings, then restart the database.

See the *Oracle Database Java Developer's Guide* for information about the `loadjava` utility and Java initialization parameters.

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, which you can accomplish with the following command (specifying *`ORACLE_HOME`* and *`J2EE_HOME`* as appropriate):

```
% 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.)

Note that JAX-RPC client proxy classes are not yet supported in Oracle9*i* or Oracle8*i*.

See the *Oracle Database Java Developer's Guide* for information about the `loadjava` utility.

Verifying or Loading the sqljutl.jar File

The following file or its contents must be loaded in the database for server-side Java invocation, such as to support Web services call-ins:

- *`ORACLE_HOME`*`/sqlj/lib/sqljutl.jar`

In Oracle Database 10*g*, its contents are preloaded in the Java VM. In Oracle9*i* or Oracle8*i*, you must load the file manually. To see if it has already been loaded, you can 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 has not already been loaded, you can use the `loadjava` utility, such as in the following example:

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

Note: To load this file, as with the file discussed in the previous section, verify that `java_pool_size` has a setting of at least 96 MB and `shared_pool_size` has a setting of at least 80 MB.

Situations for Reduced Requirements

If you will not be using certain features of JPublisher, your requirements may be less stringent:

- If you never generate classes that implement the Oracle-specific `oracle.sql.ORAData` interface (or the deprecated `oracle.sql.CustomDatum` interface), you can use a non-Oracle JDBC driver and connect to a non-Oracle database. JPublisher itself, however, must be able to connect to an Oracle database. Be aware that Oracle does not test or support configurations that use non-Oracle components. (See "Representing User-Defined SQL Types Through JPublisher" on page 1-17 for an overview of `ORAData`.)
- If you instruct JPublisher to *not* generate wrapper methods (through the setting `-methods=false`), or if your object types define no methods, then JPublisher will not generate wrapper methods or produce any SQLJ classes. Under these circumstances, there will be no SQLJ translation step, so the SQLJ translator is not required. See "Generation of Package Classes and Wrapper Methods (-methods)" on page 5-27 for information about the `-methods` option.
- 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 with JDK version 1.1 or higher. But it is advisable to upgrade to the `ORAData` interface, which requires an Oracle9*i* or higher JDBC implementation.
- 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 `utl_dbws.jar` or `utl_dbws_jserver.jar` to be loaded in the database.

JPublisher Limitations

Be aware of the following when you use JPublisher:

- There are limitations to the support for PL/SQL RECORD and indexed-by table types. First, an intermediate wrapper layer is used to map a RECORD or 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 (SQL collection). See "Type Mapping Support for PL/SQL RECORD and Indexed-by Table Types" on page 2-17 for details about how these types are supported.
- If you use an `INPUT` file to specify type mappings, 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 "INPUT File Precautions" on page 5-57 for details.
- There is a JPublisher option, `-omit_schema_names`, that has 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 enable the option (to disable the use of schema names), enter the option name, "`-omit_schema_names`", on the command line, but do *not* attempt to set "`-omit_schema_names=true`" or "`-omit_schema_names=false`". See "Omission of Schema Name from Name References (-omit_schema_names)" on page 5-28 for additional information.

What JPublisher Can Publish

The following sections describe the basic categories of publishing that the JPublisher utility supports:

- Publishing SQL User-Defined Types
- Publishing PL/SQL Packages
- Publishing Server-Side Java Classes
- Publishing SQL Queries or DML Statements
- Publishing Proxy Classes and Wrappers for Web Services Call-Outs

Publishing SQL User-Defined Types

Using JPublisher to publish SQL objects or collections as Java classes is straightforward. This section provides examples of this for the `OE` (Order Entry) schema that is part of the Oracle Database sample schema. (See *Oracle Database Sample Schemas* for detailed information.) 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 with your own.

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

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

Note: See "Declaration of Object Types and Packages to Translate (-sql)" on page 5-9 for more information about the `-sql` option.

Use the JPublisher `-user` option to specify the user name (schema name) and password. The `-sql` option specifies the types to be published. `CATEGORY_TYP` is the name of the SQL type and, separated by a colon (":"), `CategoryTyp` is the name of the corresponding Java class to be generated. JPublisher echoes to the standard output the names of the SQL types that it is publishing:

```
OE.CATEGORY_TYP
```

When you list the files in your current directory, notice that in addition to the file `CategoryTyp.java`, JPublisher has also generated the file `CategoryTypeRef.java`. This represents a strongly typed wrapper class for SQL object references to `OE.CATEGORY_TYP`. Both files are ready to be compiled with the Java compiler, `javac`.

Here is another example, for the type `CUSTOMER_TYP`, using the shorthand `-u` (followed by a space) for "`-user=`" and `-s` (followed by a space) for "`-sql=`":

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

JPublisher reports a list of SQL object types, as follows, because whenever it encounters an object type for the first time (whether through an attribute, an object reference, or a collection that has element types that themselves are objects or collections), it automatically generates a wrapper class for that type as well.

```
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: 1) source file for a Java class, such as `CustomerTyp`, to represent instances of the object type; and 2) source file for a reference class, such as `CustomerTypeRef`, to represent references to the object type. You may also have noticed the naming scheme that JPublisher uses by default: the SQL type `OE.PRODUCT_INFORMATION_TYP` turns into a Java class `ProductInformationTyp`, for example.

Even though 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 these object types.

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

Here is the JPublisher output, listing the processed types:

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

Note the following:

- If you want to unparse several types, you can list them all together in the `-sql` (`-s`) option, separated by commas, or you can supply several `-sql` options on the command line, or you can do both.
- Although JPublisher does not automatically generate wrapper classes for all subtypes, it *does* generate them for all supertypes.
- For SQL objects with methods (stored procedures), such as `CATALOG_TYP`, JPublisher uses SQLJ classes, meaning Java classes that use the SQLJ runtime during execution, 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: In Oracle9*i* or Oracle8*i* releases, the generation of SQLJ classes results in the creation of visible `.sqlj` source files. In this example, it results in `.sqlj` source files corresponding to `CATALOG_TYP` and its three subtypes. This is also true in Oracle Database 10*g* if you set the JPublisher `-compatible` flag to a value of `8i`, `both8i`, `9i`, or `sqlj`.

For any of these modes, you can use the JPublisher `-sqlj` option to translate `.sqlj` files, as an alternative to using the `sqlj` command-line utility directly.

See "Backward Compatibility Option" on page 5-43 and "Option to Access SQLJ Functionality" on page 5-42 for information about these options.

If the code that JPublisher generates does not give you 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
```

Here is the JPublisher output:

```
OE.WAREHOUSE_TYP
```

With this command, JPublisher generates both `JPubWarehouse.java` and `MyWarehouse.java`. The file `JPubWarehouse.java` is regenerated every time you rerun this command. The file `MyWarehouse.java` is created to be customized by you, and will not be overwritten by future runs of this JPublisher invocation. You can add new methods in `MyWarehouse.java`, override the method implementations from `JPubWarehouse.java`, or both. The class that is used to materialize `WAREHOUSE_TYP` instances in Java is the specialized class `MyWarehouse`. 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`*, as shown in the preceding JPublisher command, for all members of the hierarchy.

Once you have generated and compiled Java wrapper classes with JPublisher, using them is fairly straightforward. 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. (A code comment shows the corresponding `#sql` command. By default, JPublisher generates and translates the SQLJ code automatically.) See the next section, "Publishing PL/SQL Packages", for a discussion of wrapper methods and the problems posed by `OUT` and `IN OUT` arguments.

```
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 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, as 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 (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);
```

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.

The concept of representing PL/SQL stored procedures as Java methods presents a problem, however. Arguments to such functions or procedures may use the PL/SQL mode `OUT` or `IN OUT`, 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 an array `int[] abc`, for example, 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.

For additional information about the array mechanism and other mechanisms for handling `OUT` or `IN OUT` parameters, see "JPublisher Treatment of Output Parameters" on page 3-1.

Note: If your stored procedures use types that are specific to PL/SQL and are not supported by JDBC, special steps are required to map these arguments to SQL and then to Java. See "Support for PL/SQL Datatypes" on page 2-10.

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

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

Here is the JPublisher output:

```
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 all 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.

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 object type `Integer`. The following modified JPublisher invocation accomplishes this through the `-numbertypes` option.

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

Here is the JPublisher output:

```
SYS.DBMS_LOB
```

See "Mappings For Numeric Types (-numbertypes)" on page 5-18 for information about that option.

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
```

Here is the JPublisher output:

```
SCOTT.top-level_scope
```

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

Publishing Server-Side Java Classes

Oracle Database 10*g* introduces the *native Java interface*—new features for calls to server-side Java code. Previously, calling Java stored procedures and functions from a database client required JDBC calls to 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 avoid these disadvantages.

To remedy the deficiencies of JDBC calls to associated PL/SQL wrappers, the `-java` option makes convenient 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, 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 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 runtime, 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.

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

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

Use the `-java` setting in the following JPublisher command:

```
% jpub -u scott/tiger -url=jdbc:oracle:oci:@ -java=oracle.sqlj.checker.JdbcVersion
```

Note that for invocation of server-side Java, you must provide information for the database connection.

See "Declaration of Server-Side Java Classes to Translate (-java)" on page 5-7 for information about `-java` option syntax.

Publishing SQL Queries or DML Statements

The JPublisher `-sqlstatement` option enables you to publish SQL queries (`SELECT` statements) or DML statements (`INSERT`, `UPDATE`, or `DELETE` statements) as Java methods. This functionality is of potential use for Web services, but is more generally useful as well.

Specify the following through `-sqlstatement` settings:

- Java class in which the method will be published (`-sqlstatement.class=`*`classname`*)
- SQL statement and desired corresponding Java method name (`-sqlstatement.`*`methodname`*`=`*`sqlstatement`*)
- Whether JPublisher should generate a method that returns a generic `java.sql.ResultSet` instance, a method that returns an array of JavaBeans, or both methods (`-sqlstatement.return=resultset|beans|both`)

Consider the following `-sqlstatement` settings:

```
-sqlstatement.class=SqlStatement
-sqlstatement.getEmp="select ename from emp where empno=:{myno NUMBER}"
```

JPublisher generates a class named `SqlStatement` with the following method:

```
public static GetEmpRow[] getEmpBeans(int myno)
{...}
```

See "Declaration of SQL Statements to Translate (-sqlstatement)" on page 5-12 for syntax information and a complete example.

Publishing Proxy Classes and Wrappers for Web Services Call-Outs

Given a Web Services Description Language (WSDL) document at a 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 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.
2. As appropriate or necessary, creates Java wrapper classes for the Web services client proxy classes. For each proxy class that has instance methods (as is typical), a wrapper class is necessary to expose the instance methods as static methods.
3. Creates PL/SQL wrappers (call specs) for the generated classes, to make them accessible from PL/SQL.
4. Loads generated code into the database, unless you specify otherwise, and assuming you specify a database to connect to.

See "Options to Facilitate Web Services Call-Outs" on page 5-34 for details.

JPublisher Mappings and Mapping Categories

The following sections offer 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 (objects, collections, and OPAQUE 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.

Here 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. (You can also write them by hand, but this is generally not recommended.)

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.

See the next section, "Using Strongly Typed Object References for ORAData Implementations", for details about strongly typed object reference representations through the `ORAData` interface.

> **Note:** JPublisher generates classes for object reference, collection, and OPAQUE types *only* if it is generating `ORAData` classes.

- 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. (You can also write them by hand, but this is generally not recommended. Note that if you write them by hand, or if you generate classes for an inheritance hierarchy of object types, your classes must be registered using a type map.)

  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 class `oracle.sql.STRUCT` represents all object types, the class `oracle.sql.ARRAY` represents all VARRAY and nested table types, the class `oracle.sql.REF` represents all object reference types, and the class `oracle.sql.OPAQUE` 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.

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 generally modify attributes of an object or elements of a collection. The exception is that `ORAData` classes representing object reference types are not mutable, because an object reference does not have any subcomponents that could be sensibly 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 together 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. For additional information about the SQLData and ORAData interfaces, including a comparison, see the *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 (with names such as PersonRef for references to PERSON objects) are essentially 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, in the form of an instance of the corresponding Java class (or, in the case of inheritance, perhaps as 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, 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, permitting 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, PL/SQL RECORD types, and PL/SQL indexed-by table types—that are used in stored procedures or functions. (One exception is scalar PL/SQL indexed-by tables, which are currently supported in the client-side JDBC OCI driver only.) 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 also have the choice of JPublisher automatically creating a SQL object type or SQL collection type, respectively, as a middle step in the mapping.

With either mechanism, JPublisher creates PL/SQL conversion functions or uses predefined conversion functions (typically 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 generated Java code calls the wrapper function, which calls the conversion functions. Either way, only SQL types are exposed to JDBC.

See "JPublisher User Type Map and Default Type Map" on page 2-5 and "Support for PL/SQL Datatypes" on page 2-10 for additional information.

JPublisher Mapping Categories

JPublisher offers different categories of datatype mappings from SQL to Java. ("Options for Datatype Mappings" on page 5-17 describes JPublisher options to specify these mappings.)

Each type mapping option has at least two possible values: `jdbc` and `oracle`. The `-numbertypes` option has two additional alternatives: `objectjdbc` and `bigdecimal`.

The following sections describe these categories of mappings. For more information about datatype mappings, see Chapter 2.

JDBC Mapping

In JDBC mapping, most numeric datatypes are mapped to Java primitive types, such as `int` and `float`, and `DECIMAL` and `NUMBER` are mapped to `java.math.BigDecimal`. LOB types and other non-numeric built-in types are mapped to standard JDBC types, such as `java.sql.Blob` and `java.sql.Timestamp`. For object types, JPublisher generates `SQLData` classes. Because predefined datatypes 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 or method to get an attribute of a primitive type (`short`, `int`, `float`, or `double`) 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 datatypes 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 only in that it does not use primitive 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.

Object JDBC is the default mapping for numeric types.

BigDecimal Mapping

In `BigDecimal` mapping, all numeric datatypes 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. Because it uses the `oracle.sql.NUMBER` class for all numeric types, it can represent the largest numeric values that can be stored in the database.

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 Oracle JDBC 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 objects from and write objects to the database. When you deploy the generated JPublisher classes, your JDBC driver installation includes all the necessary runtime files. If JPublisher generates wrapper methods for stored procedures, then the classes that it produces use the SQLJ runtime during execution. In this case, which is typical, you must additionally have the SQLJ runtime 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 (stored procedure or function) 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, 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 WSDL document, directs the generation of appropriate client proxy classes, then generates wrapper classes, as necessary, and PL/SQL wrappers to allow Web services call-outs from PL/SQL.

The following subsections go into more detail:

- Input to JPublisher
- Output from JPublisher

In addition, see "Summary of the Publishing Process: Generation and Use of Output" on page 1-25 for a graphical representation of the flow of input and output.

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 standard output. "JPublisher Options" on page 5-1 describes all the JPublisher options.

In addition, 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. "INPUT File Structure and Syntax" on page 5-53 describes `INPUT` file syntax.

To use a properties file to specify option settings, specify the name of the properties file on the command line, 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. For more information about properties file and their formats, see "Properties File Structure and Syntax" on page 5-51.

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, and SQL queries or DML statements.

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. If, for example, a stored procedure in a PL/SQL package being published uses a SQL object type as an argument, a class will be generated to map to that SQL object type.

Java Output for User-Defined Object Types

When you run JPublisher for a user-defined object type and you 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, such as `Employee.java` for the Oracle object type `EMPLOYEE`.

- Optionally, a stub subclass, named as specified in your JPublisher settings, that you can modify as desired for custom functionality

- Optionally, an interface for the generated class or subclass to implement

- A related reference (REF) class for object references

  JPublisher generates a *`type`*`Ref.java` file for the code for the REF class associated with the object type, such as `EmployeeRef.java` 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 pre-mapped, 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.
- Advantages of using strongly typed instead of weakly typed references are described in "Using Strongly Typed Object References for ORAData Implementations" on page 1-19.

If you request `SQLData` classes instead, JPublisher does not generate the object reference class and does not generate classes for nested collection attributes or for 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 to correspond to your Oracle collection type

  For each collection type (nested table or VARRAY) it translates, JPublisher 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, 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 object elements can be materialized in Java whenever an instance of the collection is materialized.

- Optionally, an interface that is implemented by the generated type

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.

Typically, however, Java wrapper classes for SQL OPAQUE types are furnished by the provider of the OPAQUE type, such as, for example, `oracle.xdb.XMLType` for the SQL OPAQUE type `SYS.XMLTYPE`. In this case, 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 general-use server-side Java class, it creates 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, as a bridge to the corresponding PL/SQL wrappers. This is necessary to publish any proxy class instance methods as static methods, 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 (`SELECT`, `UPDATE`, `INSERT`, or `DELETE`), it creates the following:

- A Java class that implements the method that executes the SQL statement
- Optionally, a Java stub subclass, named as specified in your JPublisher settings, that you can modify as desired for custom functionality
- Optionally, a Java interface for the generated class or subclass to implement

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 specs 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 PL/SQL scripts as follows:

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

JPublisher Operation

This section discusses the basic steps in using JPublisher, summarizes the command-line syntax, and concludes with a more detailed description of a sample translation. The following topics are covered:

- Summary of the Publishing Process: Generation and Use of Output

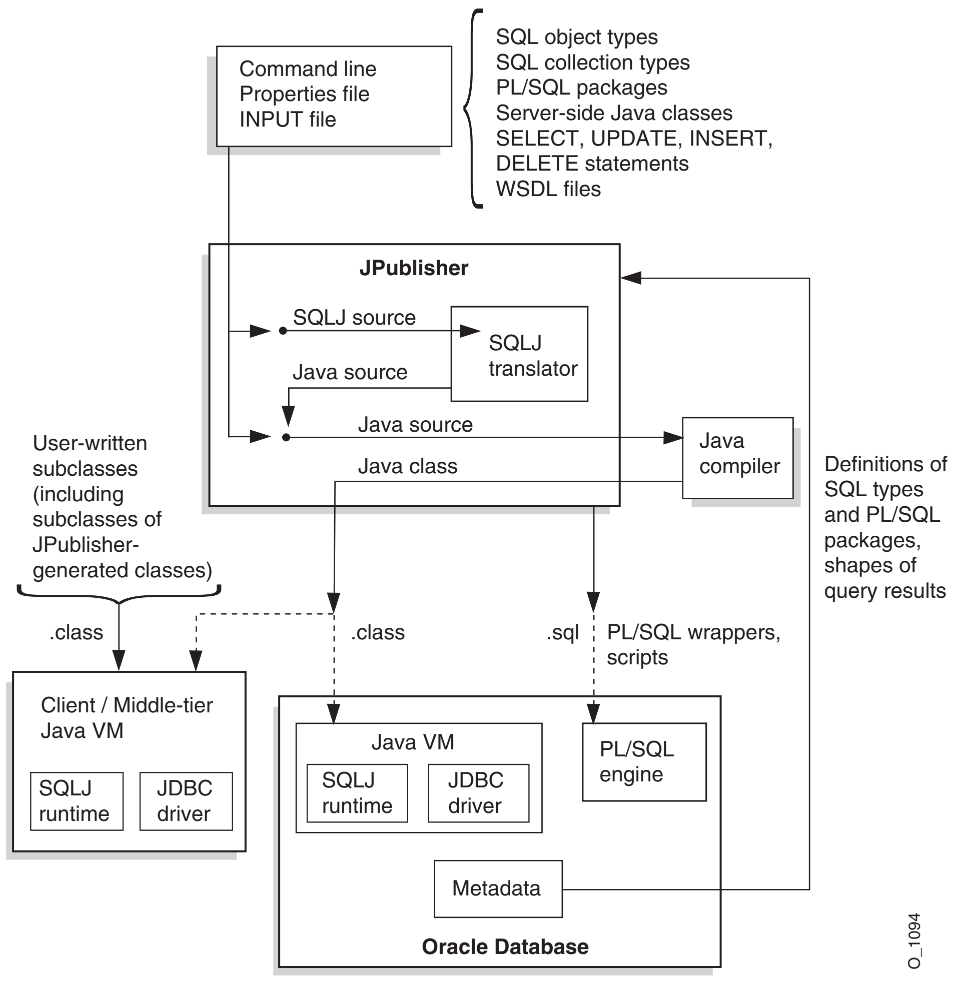
- JPublisher Command-Line Syntax
- Sample JPublisher Translation

Summary of the Publishing Process: Generation and Use of Output

This section lists the basic steps, illustrated in Figure 1–1 that follows, for publishing specified SQL types, PL/SQL packages, or server-side Java classes.

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 specified for publishing.
3. JPublisher generates `.java` or `.sqlj` source files, as appropriate, depending primarily on whether wrapper methods are created for stored procedures.
4. By default, JPublisher invokes the SQLJ translator (provided as part of the JPublisher product) to translate `.sqlj` files into `.java` files.
5. By default, the SQLJ translator (or JPublisher, for non-SQLJ classes) invokes the Java compiler to compile `.java` files into `.class` files.
6. JPublisher outputs PL/SQL wrappers and scripts (`.sql` files), 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 invoke JPublisher on the command line, typing `jpub` followed by a series of options 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, then generating one or more custom Java classes (SQLJ classes or non-SQLJ classes, as appropriate) and writing the names of the translated object types or PL/SQL packages to 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 one command line, allowing it to wrap as necessary. For clarity, this chapter refers to the input file specified by the `-input` option as the `INPUT` file (to distinguish it from any other kinds of input files).

This command directs JPublisher to connect to the database with user name `SCOTT` and password `TIGER` and to translate datatypes 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 package `corp` under the directory `demo`.

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

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

Notes:

- No spaces are permitted around equals signs (=) in a JPublisher command line.
- If you execute JPublisher without any command-line input, 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, datatype mappings, and translation later in this manual.

Note: For more examples, go to *`ORACLE_HOME`*`/sqlj/demo/jpub` in your Oracle 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 (wraparound) command line:

```
% jpub -user=scott/tiger -dir=demo -numbertypes=objectjdbc -builtintypes=jdbc
       -package=corp -case=mixed -sql=Employee
```

"JPublisher Options" on page 5-1 describes each of these options in detail.

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

Because -dir=demo and -package=corp were specified on the JPublisher command line, the translated class Employee is written to Employee.java in the following location (for a UNIX system):

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

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 setting `-serializable=true` results in the object wrapper class implementing the interface `java.io.Serializable` and in the generation of `private writeObject()` and `private readObject()` methods. The setting `-tostring=true` results in the additional 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 internal class `oracle.jpub.runtime.MutableStruct`; this instance contains the data in original SQL format. In general, you should never reference this field directly.

Instead, use the setting `-methods=always` or `-methods=named`, as necessary, to ensure that JPublisher produces `setFrom()` and `setValueFrom()` methods, then use these methods when extending a class. See "The setFrom(), setValueFrom(), and setContextFrom() Methods" on page 3-11.

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

  Additionally:
  - 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.

  (In a backward compatibility mode, this means that JPublisher generates `.sqlj` files instead of `.java` files.)
- Note that the JPublisher version provided with Oracle8*i* 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, and this is required if you are using an Oracle8*i* JDBC driver.

JPublisher also generates an `EmployeeRef.java` class. The source code is displayed here:

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

2

Datatype and Java-to-Java Type Mappings

This chapter discusses details of JPublisher support for datatype mapping, including a section on JPublisher styles and style files for Java-to-Java type mappings (primarily to support Web services). The following topics are covered:

- JPublisher Datatype Mappings
- Support for PL/SQL Datatypes
- JPublisher Styles and Style Files

JPublisher Datatype Mappings

This section discusses JPublisher functionality for mapping from SQL and PL/SQL to Java, covering the following topics:

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

Also see "Support for PL/SQL Datatypes" on page 2-10.

Overview of JPublisher Datatype Mappings

You can specify one of the following settings for datatype mappings when you use the type mapping options (`-builtintypes`, `-lobtypes`, `-numbertypes`, and `-usertypes`):

- `oracle`
- `jdbc`
- `objectjdbc` (for `-numbertypes` only)
- `bigdecimal` (for `-numbertypes` only)

These mappings, described in "JPublisher Mapping Categories" on page 1-20, affect the argument and result types JPublisher uses in the methods it generates.

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

The 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 datatypes as closely as possible so that little or no data conversion is required between the Java and the SQL formats. You do not lose any information and have greater flexibility in how you process and unpack the data. The Oracle mappings for standard SQL types are the most convenient representations if you are manipulating data within the database or moving data (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 2–1 lists the mappings from SQL and PL/SQL datatypes to Java types, using the Oracle and JDBC mappings. You can use all the supported datatypes listed in this table as argument or result types for PL/SQL methods. You can use a subset of the datatypes as object attribute types, as listed in "Allowed Object Attribute Types" on page 2-7.

The **SQL and PL/SQL Datatype** column contains all possible datatypes.

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 supplied by Oracle and are designed to minimize the overhead incurred when converting Oracle datatypes to Java types. Refer to the *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 JPublisher uses when all the type mapping options are set to `jdbc`. For standard SQL datatypes, JPublisher uses Java types specified in the JDBC specification. For SQL datatypes 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 datatypes are not directly supported by JPublisher, in particular those types that pertain to PL/SQL only. 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 2–1 SQL and PL/SQL Datatype to Oracle and JDBC Mapping Classes

SQL and PL/SQL Datatype	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 TIMESTAMP WITH TZ TIMESTAMP WITH LOCAL TZ	oracle.sql.TIMESTAMP oracle.sql.TIMESTAMPTZ oracle.sql.TIMESTAMPLTZ	java.sql.Timestamp
INTERVAL YEAR TO MONTH INTERVAL DAY TO SECOND	String (note 2)	String (note 2)
URITYPE DBURITYPE XDBURITYPE HTTPURITYPE	java.net.URL (note 3)	java.net.URL (note 3)
ROWID, UROWID	oracle.sql.ROWID	oracle.sql.ROWID
BOOLEAN	boolean (note 4)	boolean (note 4)
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 5)	Generated or predefined class (note 5)
RECORD types	Through mapping to SQL object type (note 6)	Through mapping to SQL object type (note 6)
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 2–1 (Cont.) SQL and PL/SQL Datatype to Oracle and JDBC Mapping Classes

SQL and PL/SQL Datatype	Oracle Mapping	JDBC Mapping
Indexed-by tables	Through mapping to SQL collection (note 7)	Through mapping to SQL collection (note 7)
Scalar (numeric or character) Indexed-by tables	Through mapping to Java array (note 8)	Through mapping to Java array (note 8)
User-defined subtypes	Same as for base type	Same as for base type

Datatype Mapping Notes The following notes correspond to marked 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 runtime. 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, and functions from the `SYS.SQLJUTL` package are used for the conversions. See "JPublisher User Type Map and Default Type Map" on page 2-5.
3. SQL URI types, also known as "data links", are mapped to `java.net.URL` in the JPublisher default type map, and functions from the `SYS.SQLJUTL` package are used for the conversions. See "Data Link Support and Mapping" on page 2-10.
4. 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.
5. Mapping of the SQL OPAQUE type `SYS.XMLTYPE` to the Java class `oracle.xdb.XMLType` 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 2-11.
6. 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 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; another to drop the SQL type and the PL/SQL package created by the first script. Also see "Type Mapping Support for PL/SQL RECORD and Indexed-by Table Types" on page 2-17.
7. To support a PL/SQL indexed-by table type, JPublisher first maps the indexed-by table type into a SQL collection type, 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 indexed-by table type; the other to drop the collection type and the PL/SQL package created by the first script. Also see "Type Mapping Support for PL/SQL RECORD and Indexed-by Table Types" on page 2-17.

8. If you use the JDBC OCI driver to call PL/SQL stored procedures or object methods, then you have direct support for scalar indexed-by tables, also known as PL/SQL TABLE types. In this case, use a type map entry for JPublisher that specifies the PL/SQL scalar indexed-by table type and a corresponding Java array type. JPublisher can then automatically publish PL/SQL or object method signatures that use this scalar indexed-by type. See also "Type Mapping Support for Scalar Indexed-by Tables Using JDBC OCI" on page 2-13.

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, and 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 (as appropriate). Where applicable, they also specify conversion functions to convert between PL/SQL types and SQL types. See "Options for Type Maps" on page 5-20 for additional information about syntax for the JPublisher type map commands and for documentation of the four type map options.

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 datatypes that unsupported by JDBC. (Also see "Support for PL/SQL Datatypes" on page 2-10.)
- Avoid regenerating a Java class to map to a user-defined type. For example, assume you have a user-defined `STUDENT` SQL object type 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 "Example: Using the Type Map to Avoid Regeneration", which follows shortly.

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 OPAQUE type `SYS.XMLTYPE` and the Java wrapper class `oracle.xdb.XMLType`. In addition, it maps the PL/SQL `BOOLEAN` type to Java `boolean` and to SQL `INTEGER` through two conversion functions defined in the `SYS.SQLJUTL` package. Also, the default type map provides mappings between SQL INTERVAL types and the Java `String` type, and between SQL URI types and the `java.net.URL` type.

However, you may (for example) prefer mapping the PL/SQL `BOOLEAN` type to the Java object type `Boolean` to capture SQL `NULL` values in addition to `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 This example uses the JPublisher type map to avoid having Java map classes regenerated. 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 (with the JPublisher output lines shown):

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

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

If you want to regenerate the classes for `TRIP` without regenerating the classes for `CITY`, 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, so the `City.java` and `CityRef.java` files are not regenerated. This is because of the addition of the `CITY:City` relationship to the user type map, which makes JPublisher aware that the (already existing) `City` class is to be used for mapping.

JPublisher Logical Progression for Datatype 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 specified there. See the preceding section, "JPublisher User Type Map and Default Type Map".
2. Checks the predefined Java mappings for SQL and PL/SQL types. See "SQL and PL/SQL Mappings to Oracle and JDBC Types" on page 2-2.
3. Checks whether the datatype to be mapped is a PL/SQL RECORD type or indexed-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 indexed-by table type, JPublisher generates a corresponding SQL collection type that it can

then map to Java. See "Type Mapping Support for PL/SQL RECORD and Indexed-by Table Types" on page 2-17 for details and examples.

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

Allowed Object Attribute Types

You can use a subset of the SQL datatypes in Table 2–1 as object attribute types. The allowable types 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 TIMESTAMP types `TIMESTAMP`, `TIMESTAMP WITH TIMEZONE`, and `TIMESTAMP WITH LOCAL TIMEZONE` as object attributes; however, the Oracle JDBC implementation does not.

Mapping of REF CURSOR Types and Result Sets

If a PL/SQL stored procedure or function or a SQL query returns a REF CURSOR, then JPublisher by default generates a method to map the REF CURSOR to the following:

- `java.sql.ResultSet`

In addition, for a SQL query (but not a REF CURSOR returned by a stored procedure or function), JPublisher generates a method to map the REF CURSOR to the following:

- An array of rows, in which each row is represented by a JavaBean instance

Additionally, 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 these 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`.

Take the following steps, as desired, 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 contain JPublisher and SQLJ translator classes, SQLJ runtime classes, and JDBC classes, respectively.
2. For mapping to `Source`, use JDK 1.4. (This class is not defined in earlier JDK versions.)
3. For mapping to `OracleWebRowSet`, add *`ORACLE_HOME`*`/jdbc/lib/ocrs12.jar` to the classpath.
4. For mapping to `Document`, add *`ORACLE_HOME`*`/lib/xmlparsev2.jar` to the classpath.

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 CURSORs. Following 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 you copy this file into `myrefcursormaps.properties`. There are four `MAPPING` sections, intended to map REF CURSORs to `ResultSet`, `OracleWebRowSet`, `Document`, and `Source` (according to the `SOURCETYPE` and `TARGETTYPE` entries). For this example, lines are commented out (by "#" characters) 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 as follows to use your custom mappings:

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

Data Link Support and Mapping

JPublisher supports the use of SQL URI types that store universal resource locators (URIs), referred to as *data links*. These types—`SYS.URITYPE` and the subtypes `SYS.DBURITYPE`, `SYS.XDBURITYPE`, and `SYS.HTTPURITYPE`—are mapped to the `java.net.URL` Java type.

As an example, consider the following SQL type that uses a `URITYPE` attribute:

```
create or replace type dl_obj as object (myurl sys.uritype);
/
```

And assume that JPublisher is invoked with the following command line (with JPublisher output also shown):

```
% jpub -u scott/tiger -s dl_obj
SCOTT.DL_OBJ
```

This command results in the following methods in the generated Java code:

```
public java.net.URL getMyurl() throws SQLException;
public void setMyurl(java.net.URL myurl) throws SQLException;
```

JPublisher adds the following definitions to the `jpub.properties` file to specify the URI type mappings:

```
jpub.adddefaulttypemap=
SYS.URITYPE:java.net.URL:VARCHAR2:SYS.URIFACTORY.GETURI:SYS.SQLJUTL.URI2CHAR
jpub.adddefaulttypemap=
SYS.DBURITYPE:java.net.URL:VARCHAR2:SYS.DBURITYPE.CREATEURI:SYS.SQLJUTL.URI2URL
jpub.adddefaulttypemap=
SYS.XDBURITYPE:java.net.URL:VARCHAR2:SYS.XDBURITYPE.CREATEURI:SYS.SQLJUTL.URI2URL
jpub.adddefaulttypemap=
SYS.HTTPURITYPE:java.net.URL:VARCHAR2:SYS.HTTPURITYPE:SYS.SQLJUTL.URI2URL
```

This includes specification of data conversion functions. Also see "Type Mapping Support Through PL/SQL Conversion Functions" on page 2-15 and "Options for Type Maps" on page 5-20.

Support for PL/SQL Datatypes

There are three scenarios if JPublisher encounters a PL/SQL stored procedure or stored function (including methods of SQL object types) that 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 "JPublisher User Type Map and Default Type Map" on page 2-5.
- If there is no mapping in the type maps, and the PL/SQL type is a RECORD type or indexed-by table type, then JPublisher generates a corresponding SQL type that JDBC supports. For PL/SQL RECORD types, JPublisher generates a SQL object type to bridge between the RECORD type and Java. For indexed-by table types, 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
- Type Mapping Support for Scalar Indexed-by Tables Using JDBC OCI
- Type Mapping Support Through PL/SQL Conversion Functions
- Type Mapping Support for PL/SQL RECORD and Indexed-by Table Types
- Direct Use of PL/SQL Conversion Functions Versus Use of Wrapper Functions
- Other Alternatives for Datatypes Unsupported by JDBC

Type Mapping Support for OPAQUE Types

This section describes JPublisher type mapping support for OPAQUE types in general and the OPAQUE type `SYS.XMLTYPE` in particular.

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

General 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 to that type map:

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

In Oracle Database 10*g*, the SQL OPAQUE type `SYS.XMLTYPE` is mapped to the Java class `oracle.xdb.XMLType` through the JPublisher default type map. (Also see the next section, "Support for XMLTYPE".) 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 attribute `xml` is published using `oracle.xdb.XMLType`, 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 option `-defaulttypemap` is for setting the JPublisher default type map. Giving it no value, as in the preceding example, clears it.

Note: See "JPublisher User Type Map and Default Type Map" on page 2-5 for information about JPublisher type maps. See "Options for Type Maps" on page 5-20 for information about `-defaulttypemap`, `-addtypemap`, and the other type map options.

Support for XMLTYPE

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

The SQLJ runtime 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 out of a `String` instance.

For Java-to-Java type transformations (often necessary for Web services), the style file `webservices-common.properties` specifies the preceding mapping, as well as the Java-to-Java mapping of `SimpleXMLType` to `java.lang.String`. Therefore, 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 "JPublisher-Generated Subclasses for Java-to-Java Type Transformations" on page 3-16 for information about how this process works. See "JPublisher Styles and Style Files" on page 2-22 for general information about style files.

The style files `webservices9.properties` and `webservices10.properties` include `webservices-common.properties`. 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; the `webservices10.properties` file maps it to `javax.xml.transform.Source`.

Take 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 using `org.w3c.dom.DocumentFragment`. Following is an example that contains the JPublisher command line and portions of the PL/SQL procedure, the Java interface, the base Java class, and the user subclass.

Here is the JPublisher command line:

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

These are the SQL definitions:

```
procedure setXMLMessage(x xmltype, y number);
function getXMLMessage(id number) return xmltype;
```

Here are the definitions in the `XTest` interface:

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

These are the definitions in `XTestBase.java`:

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

Following are the definitions in `XTestUser.java`:

```
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 Indexed-by Tables Using JDBC OCI

The Oracle JDBC OCI driver directly supports PL/SQL scalar indexed-by tables with numeric or character elements. (If you are not using the JDBC OCI driver, see "Type Mapping Support for PL/SQL RECORD and Indexed-by Table Types" on page 2-17.) An indexed-by table with numeric elements can be mapped to the following Java array types:

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

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

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

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

- Whenever you use the indexed-by table type in an `OUT` or `IN OUT` parameter position, you must specify the maximum number of elements. (This is optional otherwise.) This is defined 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 indexed-by tables with character elements, you can optionally specify the maximum size of an individual element (in bytes). This setting is defined using SQL-like size syntax. For example, for an indexed-by table used for `IN` arguments, you could specify `String[](30)`. Or specify `oracle.sql.CHAR[20](255)` for an indexed-by table of maximum length 20, each of whose elements will not exceed 255 bytes.

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

As an example, consider the following code fragment from the definition of a PL/SQL package `INDEXBY` in the schema `SCOTT`. Assume this is available in a file `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 indexed-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 **(**...**)**. Or 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 "Properties File Structure and Syntax" on page 5-51 for information about properties files. For general information about the `-addtypemap` option, see "Additional Entry to the User Type Map (-addtypemap)" on page 5-21.

Additionally, as a convenience feature, 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 indexed-by tables only with the JDBC OCI driver. 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 indexed-by table types, as well as defining conversion functions that map between the two. Refer to "Type Mapping Support for PL/SQL RECORD and Indexed-by Table Types" on page 2-17.

Type Mapping Support Through PL/SQL Conversion Functions

This section discusses the general mechanism 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. Sections that follow this section are concerned with mapping issues specific to PL/SQL RECORD types and PL/SQL indexed-by table types, respectively.

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. (Although one exception is scalar indexed-by tables. See the preceding section, "Type Mapping Support for Scalar Indexed-by Tables Using JDBC OCI".)

JPublisher makes this task more convenient through use of its type maps. (Also see "JPublisher User Type Map and Default Type Map" on page 2-5.) 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 following 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, typically generated by JPublisher, that implements the `oracle.sql.ORAData` or `java.sql.SQLData` interface.

- Name of a conversion function that maps the SQL type to the PL/SQL type
- Name of a 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
```

Also see "Options for Type Maps" on page 5-20.

As an example, consider a type map entry for supporting the PL/SQL type `BOOLEAN`. 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, `INT2BOOL`, that maps from SQL to PL/SQL (from `NUMBER` to `BOOLEAN`)

  Here is the code for that function:

  ```
  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, `BOOL2INT`, that maps from PL/SQL to SQL (from `BOOLEAN` to `NUMBER`)

  Here is the code for that function:

  ```
  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;
  ```

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. See "JPublisher User Type Map and Default Type Map" on page 2-5. 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 (where this is a single wraparound command line):

```
% 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 in the discussion of RECORD types and indexed-by table types later in this chapter, 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, calling this original stored procedure and processing 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. Also see "Direct Use of PL/SQL Conversion Functions Versus Use of Wrapper Functions" on page 2-20 and "PL/SQL Code Generation Options" on page 5-31.

- `-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 from each unsupported PL/SQL type to the corresponding SQL type, and a function to convert from each corresponding SQL type back 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 specify no package name, 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 creates the PL/SQL package. The dropper script drops these SQL types and the PL/SQL package.

  If the files already exist, they will be overwritten. If no file names are specified, JPublisher will write to 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 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 *`flag`* setting can be any of the following.

  – `true` (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 if PL/SQL types are used only as `IN` parameters or for the function return, then generated code instead calls the stored procedure directly, processing its PL/SQL input or output through the appropriate conversion functions.

  If a PL/SQL type is used as an `OUT` or `IN OUT` parameter, 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 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 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.

Type Mapping Support for PL/SQL RECORD and Indexed-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 indexed-by table types (also known as PL/SQL TABLE types). This is the only way that a RECORD type or indexed-by table type can be published; there is no way to explicitly request any such types to be published through JPublisher option settings.

Notes:

- There are limitations to the JPublisher support described here for PL/SQL RECORD and indexed-by table types. First, as covered in detail later, an intermediate wrapper layer is required to map a RECORD or indexed-by-table argument to a SQL type that JDBC can support. 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 (SQL collection).
- If you are using the JDBC OCI driver and require only the publishing of scalar indexed-by tables, you can use the direct mapping between Java and these types outlined in "Type Mapping Support for Scalar Indexed-by Tables Using JDBC OCI" on page 2-13.

The following sections demonstrate JPublisher support for PL/SQL RECORD types and indexed-by table types, respectively:

- Sample Package for RECORD Type and Indexed-by Table Type Support
- Support for RECORD Types
- Support for Indexed-by Table Types

Sample Package for RECORD Type and Indexed-by Table Type Support

The following PL/SQL package is used to illustrate JPublisher support for PL/SQL RECORD and indexed-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;
```

The package defines a PL/SQL RECORD type, `EMP_REC`, and a PL/SQL indexed-by table type, `EMP_TBL`. Use the following command (a single wraparound command line) to publish the `COMPANY` package. The JPublisher output is also shown:

```
% jpub -u scott/tiger -s COMPANY:Company -plsqlpackage=WRAPPER1
  -plsqlfile=wrapper1.sql,dropper1.sql
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 the preceding example, JPublisher generates `Company.java` (a SQLJ class) 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 indexed-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 indexed-by table type

Support for RECORD Types

This section continues the example in the preceding section, "Sample Package for RECORD Type and Indexed-by Table Type Support". For the PL/SQL RECORD type `EMP_REC`, JPublisher generates the corresponding SQL object type `COMPANY_EMP_REC`. 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 SQL object type `COMPANY_EMP_REC` into the Java source file `CompanyEmpRec.java`.

Once the PL/SQL RECORD type is published, you can add the mapping to the type map. Here 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 `EMP_REC` RECORD type. For example, the following JPublisher invocation uses `done.properties` with this type map entry (using the `-u` shorthand for `-user` and `-p` for `-props`). JPublisher output is also shown:

```
% jpub -u scott/tiger -p done.properties -s COMPANY -plsqlpackage=WRAPPER2
       -plsqlfile=wrapper2.sql,dropper2.sql
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 Indexed-by Table Types

This section continues the example begun in "Sample Package for RECORD Type and Indexed-by Table Type Support" on page 2-18, examining support for the indexed-by table type `EMP_TBL` with elements of type `EMP_REC`.

To support an indexed-by table type, a SQL collection type must be defined that permits conversion to and from the PL/SQL indexed-by table type. JPublisher also supports PL/SQL nested tables and VARRAYs in the same fashion. Therefore, JPublisher generates essentially 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 indexed-by table type `EMP_TBL`, JPublisher generates a SQL collection type, and conversion functions between the indexed-by table type and the SQL collection type.

Here is what JPublisher generates, in addition to what what shown for the RECORD type earlier:

```
-- 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 indexed-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 indexed-by table type.

For example, if you add the following entry into a properties file that you use in invoking JPublisher (`done.properties`, for example), then JPublisher will use the provided type map and avoid republishing that indexed-by table type:

```
jpub.addtypemap=SCOTT.COMPANY.EMP_TBL:CompanyEmpTbl:COMPANY_EMP_TBL:
WRAPPER1.COMPANY_EMP_TBL2PL:WRAPPER1.PL2COMPANY_EMP_TBL
```

(Use of the type map to avoid republishing is also discussed in "JPublisher User Type Map and Default Type Map" on page 2-5.)

Direct Use of PL/SQL Conversion Functions Versus Use of Wrapper Functions

The preceding sections, beginning with "Type Mapping Support Through PL/SQL Conversion Functions" on page 2-15, discuss how JPublisher uses PL/SQL conversion functions to convert between PL/SQL types, which are generally not supported by JDBC, and corresponding SQL types that have been defined. In generating Java code to invoke a stored procedure that uses a PL/SQL type, JPublisher can employ either of the following modes of operation:

- Invoke the stored procedure directly, processing its 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, 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. Also see "Generation of PL/SQL Wrapper Functions (-plsqlmap)" on page 5-32.

As an example, consider the PL/SQL stored procedure SCOTT.COMPANY.GET_EMP_TBL that returns the PL/SQL indexed-by table type EMP_TBL. Assume that the COMPANY package, introduced in "Sample Package for RECORD Type and Indexed-by Table Type Support" on page 2-18, is processed by JPublisher through the following command (with JPublisher output also shown):

```
% jpub -u scott/tiger -s COMPANY:Company -plsqlpackage=WRAPPER1
       -plsqlfile=wrapper1.sql,dropper1.sql -plsqlmap=false
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 indexed-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 indexed-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, WRAPPER1.GET_EMP_TBL in this example, looks like the following, calling the original stored procedure and processing its output through the conversion function:

```
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 looks like this:

```
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, for any stored procedure for which the Java code for direct invocation and conversion is too complex, or for 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 Datatypes Unsupported by JDBC

The preceding sections describe the mechanisms that JPublisher employs to access PL/SQL types unsupported in 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, for example, 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, but the data can be used if converted to a type, such as `java.lang.String`, that is supported by Web services.

Typically, style files are provided by Oracle, but there may be situations in which you would want to edit or create your own.

The following sections discuss features and usage of styles and style files:

- Style File Specification and Locations
- Style File Formats
- Summary of Key Java-to-Java Type Mappings in Oracle Style Files
- Use of Multiple Style Files

Note: JPublisher must generate user subclasses to implement its use of style files and Java-to-Java type transformations. Also see "JPublisher-Generated Subclasses for Java-to-Java Type Transformations" on page 3-16.

Style File Specification and Locations

Use the JPublisher `-style` option, also discussed in "Style File for Java-to-Java Type Mappings (-style)" on page 5-20, 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, using the first file that it finds:

1. First, it looks for the following resource in the classpath:

   ```
   /oracle/jpub/mesg/stylename.properties
   ```

2. Next, it takes `stylename` as a (possibly qualified) resource name and looks for the following in the classpath:

   ```
   /stylename-dir/stylename-base.properties
   ```

3. Finally, it takes `stylename` as a (possibly qualified) name 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.

As an example of scenario #1, 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 setting `-style=webservices` uses `/oracle/jpub/mesg/webservices.properties` from `translator.jar` (even if there is also a `webservices.properties` file in the current directory).

However, if you specify `-style=mystyle`, and no `mystyle.properties` resource is 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 in your classpath. Each file maps Oracle JDBC types into 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 Formats

The key portion of a style file is the `TRANSFORMATION` section—everything between the `TRANSFORMATION` tag and `END_TRANSFORMATION` tag. This section describes type transformations (Java-to-Java mappings) to be applied to types used for object attributes or in method signatures.

For convenience, there is also an `OPTIONS` section in which you can specify any other JPublisher option settings. In this way, a style file can replace the functionality of any other JPublisher properties file, in addition to specifying mappings.

Note: The following details about style files are provided for general information only, and are subject to change. Typically you will not write style files yourself, instead using files provided by Oracle.

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 (from a `MAPPING` tag to an `END_MAPPING` tag) for each mapping you specify. Each `MAPPING` section includes a number of subtags with additional information. `SOURCETYPE` and `TARGETTYPE` subtags are required. Within each `TARGETTYPE` section, you should generally provide information for at least the `RETURN`, `IN`, and `OUT` cases, using the corresponding tags. (See the comments for these tags in the template.)

```
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 "Passing Output Parameters in JAX-RPC Holders" on page 3-3 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:

```
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 mappings of REF CURSORs. These mappings are summarized in Table 2–2.

Table 2–2 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 their own mappings. For `SimpleXMLType`, note that `DocumentFragment` overrides `String` if you set `-style=webservices9`, and `Source` overrides `String` if you set `-style=webservices10`.

See "Mapping of REF CURSOR Types and Result Sets" on page 2-7 for more information about how to use some or all of the mappings for result sets.

Use of Multiple Style Files

JPublisher allows multiple `-style` options in the command line, with the following behavior:

- The `OPTIONS` sections are concatenated.
- The `TRANSFORMATION` sections are concatenated, except entries in `MAPPING` subsections are overridden as applicable. A `MAPPING` entry from a style file specified later in the command line overrides a `MAPPING` entry with the same `SOURCETYPE` specification from a style file specified earlier in the command line.

This functionality is useful if you want to overwrite earlier defined type mappings or add new type mappings. For example, if you want to map `SYS.XMLTYPE` into `java.lang.String`, you can append the setting `-style=xml2string` to the JPublisher command line, assuming for this example that this will access the style file `./xml2string.properties`, which 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` into `oracle.xdb.XMLType`, as discussed in "Type Mapping Support for OPAQUE Types" on page 2-11.

---

3 Generated Classes and Interfaces

This chapter discusses details and concepts of the classes, interfaces, and subclasses that JPublisher generates, including how output parameters are treated (PL/SQL `IN OUT` or `OUT` parameters), how overloaded methods are translated, and how to use the generated classes and interfaces. The following topics are covered:

JPublisher Treatment of Output Parameters

Stored procedures called through JDBC do not have the same parameter-passing behavior as ordinary Java methods. This affects the code 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.

By contrast, 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, copies of the modified parameters are 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 three possible parameter modes: `IN`, `OUT`, or `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 three alternatives for holders that handle PL/SQL `OUT` or `IN OUT` parameters:

- Arrays
- 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 "Holder Types for Output Arguments (-outarguments)" on page 5-29 for syntax information.

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

The simplest 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:

- You assign the "before" value of the parameter to element [0] of an array.
- You pass the array to your wrapper method.
- The wrapper method assigns the "after" value of the parameter to element [0] of the array.
- After executing the method, you extract the "after" value from the array.

A setting of -outarguments=array (the default) instructs JPublisher to use this single-element array mechanism to publish any OUT or IN OUT argument.

Here is an example:

```
Person [] pa = {p};
x.f(pa);
p = pa[0];
```

Assume that x is an instance of a JPublisher-generated class that has the method f(), which is a wrapper method for a stored procedure that uses a SQL PERSON object as an IN OUT parameter. The type PERSON maps to the Java class Person; p is a Person instance; and pa[] is a single-element Person array.

The array technique 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, 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 `this`, 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:

- 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 (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 (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` function-return as a Java string, and returns 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, 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, in the `HOLDER` subtag inside the `TARGETTYPE` section for the appropriate mapping. If no holder class is specified, then JPublisher chooses one according to defaults. See "JPublisher Styles and Style Files" on page 2-22 for details about style files.

For general information about JAX-RPC and holders, see the *Java API for XML-based RPC, JAX-RPC 1.0* specification, available at the following location:

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

As an example, again 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=holder`, the following is an example of how the function is published. 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`.

Assume the following JPublisher command to publish the function `g`. (The `webservices10` style file contains an entry for `-outarguments=holder`.)

```
% jpub -u scott/tiger  -s toplevel"(g)":ToplevelG -style=webservices10
```

Here is the published interface:

```
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;
```

Note: See "JPublisher-Generated Subclasses for Java-to-Java Type Transformations" on page 3-16 for further discussion of how JPublisher uses style files and holder classes for Java-to-Java type transformations for PL/SQL output arguments.

Passing Output Parameters in Function Returns

You can use the setting `-outarguments=return` as a workaround for supporting method signatures in Web services that do not use JAX-RPC holder types or arrays. In a situation in which there is no support for JAX-RPC holders, the `-outarguments=return` setting allows `OUT` or `IN OUT` data to be returned in function results.

Once again, 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 (a wraparound command, with output also shown) to publish the function `g`. Although the `webservices10` style file specifies `-outarguments=holder`, the `-outarguments=return` setting comes after the `-style` setting so takes precedence.

```
% jpub -u scott/tiger  -s toplevel"(g)":ToplevelG -style=webservices10
      -outarguments=return
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 Java class `ToplevelGUser_g_Out` to support output values of the function `g` through return data.

Notes:

- The "`_g_Out`" appended to the user class name is according to the JPublisher naming convention 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; the "`_Out`" reflects the `OUT` modifier in the corresponding PL/SQL call spec. So `ToplevelGUser_g_Out` is the Java type created for the output data of the `g()` method in class `ToplevelGUser`. (The user class name is according to the naming pattern specified through the `webservices10` style file.)
- Normally, 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 return type `ToplevelGUser_g_Out` 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. Overloaded methods with different signatures in PL/SQL may have identical signatures in Java, however, 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 "Style File Specification and Locations" on page 2-22 for information about `-style` settings and properties files.)

JPublisher solves such naming conflicts by appending the first letter of the return type (as applicable) and the first letter of each argument type (as applicable) to the method name. If conflicts still remain, 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 into 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, for example), 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 "Generation of Package Classes and Wrapper Methods (-methods)" on page 5-27 for additional information about the `-methods` option.

JPublisher Generation of SQLJ Classes

Note: The term *SQLJ classes* refers to Java classes that call the Oracle SQLJ runtime APIs. By default in Oracle Database 10*g*, `.sqlj` source files are invisible to the user, being translated automatically and deleted. (If desired, however, you can use settings of the JPublisher `-compatible` option to generate visible `.sqlj` files, skipping the automatic translation.)

Also see "JPublisher Usage of the Oracle SQLJ Implementation" on page 1-4 and "Backward Compatibility Option" on page 5-43.

When `-methods=all` (the default) or `-methods=true`, JPublisher typically generates SQLJ classes for PL/SQL packages and for object types, using both `ORAData` and `SQLData` implementations. The 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 runtime.

SQLJ classes include wrapper methods that invoke the server methods (stored procedures) of object types and packages. This section describes how to use these classes.

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 one (or more) of the types contains stored procedures, then by default JPublisher generates SQLJ classes for all the SQL types, not just the types that have stored procedures. (But 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. You can avoid this scenario, however, 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.
  - Explicitly associate the wrapper class instance with a SQLJ connection context instance through the `setConnectionContext()` method.
  - Implicitly use the static SQLJ default connection context instance for the wrapper class instance. This occurs if you do not supply any connection information.

  See "More About Connection Contexts and Instances in SQLJ Classes" on page 3-9 for additional information.
- In Oracle8*i* compatibility mode, instead of the constructor taking a `DefaultContext` instance or user-specified-class instance, 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 That JPublisher Generates 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 ran JPublisher); one 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.

Important: See the preceding section, "Important Notes About Generation of SQLJ Classes".

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 will 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 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 That JPublisher Generates 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, or by retrieving the SQL object through JDBC calls that you write (or through SQLJ `#sql` statements, if you are in a backward compatibility mode and using SQLJ source files directly).
- Construct the Java object with the no-argument constructor and set its attributes using the `set`*`XXX`*`()` methods, or construct the Java object with the constructor that accepts values for all the object attributes. Typically, you will subsequently use the `setConnection()` or `setConnectionContext()` method to associate the object with a database connection before invoking any of its wrapper methods. If you do not explicitly associate the object with a JDBC or SQLJ connection before invoking 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 ran 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.

Important: See "Important Notes About Generation of SQLJ Classes" on page 3-7.

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 (or in SQLJ `#sql` statements if you are in a backward compatibility mode and using SQLJ source files directly).

There is a Java attribute for each attribute of the corresponding SQL object type, with `get`*`XXX`*`()` and `set`*`XXX`*`()` accessor methods for each attribute. The accessor method names are of the form `getFoo()` and `setFoo()` for attribute `foo`. JPublisher does not generate fields for the attributes.

By default, the class includes wrapper methods that invoke the associated Oracle object methods (stored procedures) executing in the server. The wrapper methods are all instance methods, whether or not the server methods are. 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. These methods are specified in the `ORAData` and `ORADataFactory` interfaces:

- `create()`
- `toDatum()`

These methods are not generally intended for your direct use. In addition, JPublisher generates the methods `setFrom(`*`otherObject`*`)`, `setValueFrom(`*`otherObject`*`)`, and `setContextFrom(`*`otherObject`*`)` that you can use to copy the value or connection information from one object instance to another.

More About Connection Contexts and Instances in SQLJ Classes

Note: "Connection context" is a SQLJ term regarding database connections. For those unfamiliar with SQLJ, the term is briefly defined in "JPublisher Usage of the Oracle SQLJ Implementation" on page 1-4.

The class that JPublisher uses in creating SQLJ connection context instances depends on how you set the `-context` option when you run JPublisher, as follows:

- A setting of `-context=DefaultContext` (the default) results in JPublisher using instances of the standard `sqlj.runtime.ref.DefaultContext` class.
- A setting of a user-defined class that is in the classpath and that implements the standard `sqlj.runtime.ConnectionContext` interface results in JPublisher using instances of that class.

- A setting of `-context=generated` results in declaration of the `static` connection context class `_Ctx` in the JPublisher-generated class. JPublisher uses instances of this class for connection context instances. This is appropriate for Oracle8*i* compatibility mode, but generally not recommended otherwise.

See "SQLJ Connection Context Classes (-context)" on page 5-15 for more information about the `-context` option.

Note: It is no longer routine (as it was in the Oracle8*i* version) for JPublisher to declare a connection context instance `_ctx`. This is used in Oracle8*i* compatibility mode, however (`-compatible=8i` or `-compatible=both8i`), with `_ctx` being declared as a `protected` instance of the static connection context class `_Ctx`.

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. See the following discussion for more information about these methods.

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. (This is not necessary if you have already specified a connection context instance through the constructor.)

  This 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, such as object reference types or BLOBs, that are valid only for a particular connection.

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 generated SQLJ classes. If necessary, you can use the setting `-methods=always` to ensure that SQLJ classes are produced. See "Generation of Package Classes and Wrapper Methods (-methods)" on page 5-27.

- 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 conclusion of processing, use the `release()` method to free resources in the SQLJ runtime. This prevents a possible memory leak.

The setFrom(), setValueFrom(), and setContextFrom() Methods

JPublisher provides the following utility methods in 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 following:

```
x.setFrom(y);
```

and:

```
x.setValueFrom(y);
x.setContextFrom(y);
```

JPublisher Generation of Non-SQLJ Classes

When `-methods=false`, 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 runtime during execution, so JPublisher generates what is referred to as *non-SQLJ classes*, meaning classes that do not call the SQLJ runtime APIs. All this is true regardless of whether you use an `ORAData` implementation or a `SQLData` implementation.

Notes:

- When `-methods=false`, 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` setting is `false` or `true`.

To use an instance of a class that JPublisher generates for an object type when `-methods=false`—or for a reference, VARRAY, or nested table type—you must first initialize the object.

Similarly to the case with JPublisher-generated SQLJ classes, 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, or by retrieving the SQL object through JDBC calls that you write (or through SQLJ `#sql` statements, if you are in a backward compatibility mode and using SQLJ source files directly).
- 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, such as a reference or a `BLOB`, that is valid only for a particular connection.

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, 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 (or in SQLJ `#sql` statements if you are in a backward compatibility mode and using SQLJ source files directly).
- 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.
  - 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. Then re-insert the Java array into the database if you want to update the data there.

- You cannot modify an object reference, because it is an immutable entity; however, you can read and write the SQL object it references, using the `getValue()` and `setValue()` methods.

  The `getValue()` method returns a copy of the SQL object to which the reference refers. The `setValue()` method updates a SQL object type instance in the database, taking as input an instance of the Java class that represents the object type. 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.

A few methods have not been mentioned yet. You can use the `getORADataFactory()` method in JDBC code to return an `ORADataFactory` object. You can pass this `ORADataFactory` object to the `getORAData()` method in the classes `ArrayDataResultSet`, `OracleCallableStatement`, and `OracleResultSet` 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 VARRAYs and nested tables have methods that implement features of the `oracle.sql.ARRAY` class:

- `getBaseTypeName()`
- `getBaseType()`
- `getDescriptor()`

JPublisher-generated classes for VARRAYs and nested tables do not, however, extend `oracle.sql.ARRAY`.

With Oracle mapping, JPublisher generates the following methods for the Oracle JDBC driver to use. These methods are specified in the `ORAData` and `ORADataFactory` interfaces:

- `create()`
- `toDatum()`

These methods are not generally intended for your direct use; however, you may want to use them if converting from one object reference Java wrapper type to another.

JPublisher 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.

"Publishing SQL User-Defined Types" on page 1-11 and "Publishing PL/SQL Packages" on page 1-14 discuss how to use the JPublisher `-sql` option to publish user-defined types and PL/SQL packages.

In addition, 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.

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 "JPublisher Styles and Style Files" on page 2-22 for related information.

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 at some future time to regenerate the class. If you regenerate a class that you have modified in this way, then your changes (that is, 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 then customize as desired. Wherever the SQL type is referenced in code (such as where it is used as an argument), the SQL type will be mapped into the subclass, rather than into 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 in order to use a Java type that is suitable.

These topics are covered in the following sections:

- Extending JPublisher-Generated Classes
- JPublisher-Generated Subclasses for Java-to-Java Type Transformations

Extending JPublisher-Generated Classes

Suppose you want JPublisher to generate the class `JAddress` from the SQL object type `ADDRESS`. 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 must alter the code it generates in the following ways:

- It generates the reference class `MyAddressRef` rather than `JAddressRef`.

- It uses the `MyAddress` class instead of the `JAddress` class to represent attributes whose SQL type is `ADDRESS`, or to represent VARRAY and nested table elements whose SQL type is `ADDRESS`.
- It uses the `MyAddress` factory instead of the `JAddress` factory when the `ORADataFactory` interface is used to construct Java objects whose SQL type is `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. If the source file for the `MyAddress` class already exists, however, it is left untouched by JPublisher.

Note: For information about changes between Oracle8*i* and Oracle9*i* for user-written subclasses of classes generated by JPublisher, see "Changes in JPublisher Behavior Between Oracle8i and Oracle9i" on page 4-5.

Syntax for Mapping to Alternative Classes

JPublisher has 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, this is:

```
-sql=ADDRESS:JAddress:MyAddress
```

See "Declaration of Object Types and Packages to Translate (-sql)" on page 5-9 for information about the `-sql` option.

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
```

See "INPUT File Structure and Syntax" on page 5-53 for information about the `INPUT` file.

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, this attribute is created as an instance of `MyAddress`. Or, if you retrieve an `ADDRESS` object directly, it is retrieved into an instance of `MyAddress`.

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—generated in `MyAddress.java` in the preceding example—is where you place your custom code.

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 `ORADataFactory` `create()` method such as the following.

  ```
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
     return create(new UserClass(),d,sqlType);
  }
  ```

  When the class is part of an inheritance hierarchy, however, 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, described in "JPublisher Styles and Style Files" on page 2-22, 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, such as `java.lang.String`, that is supported by Web services.

If you use the JPublisher `-style` option to specify a style file, JPublisher generates subclasses that implement the Java-to-Java type mappings specified in the style file. This includes the use of "holder" classes, introduced in "Passing Output Parameters in JAX-RPC Holders" on page 3-3, for handling output arguments—data corresponding to PL/SQL `OUT` or `IN OUT` types.

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. (This style file is supplied by Oracle and is typically appropriate for using Web services in an Oracle Database 10*g* environment.) The `webservices10.properties` file specifies the following (among other things):

- The mapping of the Java type `oracle.sql.SimpleXMLType` (which is not supported by Web services) to the Java type `javax.xml.transform.Source` (which is):

  ```
  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 type `XMLTYPE`.

- The inclusion of `webservices-common.properties`:

  ```
  INCLUDE webservices-common
  ```

The `webservices-common.properties` file (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 generation of the interface `FooPack`, the base class `FooPackBase`, and the user subclass `FooPackUser`, which extends `FooPackBase` and implements `FooPack`. See "Class and Interface Naming Pattern (-genpattern)" on page 5-25 for general information about the `-genpattern` option.

- The mapping of the Java type `oracle.sql.CLOB` (which is not supported by Web services) to the Java type `java.lang.String` (which is):

  ```
  SOURCETYPE oracle.sql.CLOB
  TARGETTYPE java.lang.String
  ...
  ```

Recall the calling sequence for the `foo` stored function:

```
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 support the output argument. (See "Passing Output Parameters in Arrays" on page 3-2.)

The user subclass has the following corresponding method declaration:

```
public java.lang.String foo (SourceHolder _xa_inout_x, Integer b);
```

This 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` (all as described earlier).

Following is the class `SourceHolder`, the holder class for `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 the example, this is as follows in the `foo()` method of the class `FooPackUser`:

```
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 (data input).
2. After processing (which occurs inside the type conversion layer), it assigns the `SourceHolder` object from the `Source` object that was retrieved and processed (data output).

The type conversion layer first takes the target type (`TARGETTYPE` from the style file), next converts it to the source type (`SOURCETYPE` from the style file), then calls the corresponding method in the base class (which uses JDBC to invoke the wrapped stored function), and finally 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 (data input).
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 (output from the `foo` stored function).
5. It converts the `SimpleXMLType` object back to a `Source` object for the holder layer (data output).

Note: See "Generated Code: User Subclass for Java-to-Java Transformations" on page A-1 for the complete code.

JPublisher Support for Inheritance

This section primarily discusses inheritance support for `ORAData` types, explaining 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)

Following this information is a brief overview of standard inheritance support for `SQLData` types, with reference to appropriate documentation for further information.

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 (
...
);
```

And consider the following JPublisher command line to create corresponding Java classes (a single wraparound command):

```
% 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 the 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.

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 a minimum, the leaf classes—before using these mapped types in your code. For example:

```
new Instructor(); // required
new Student();    // required
new Person();     // optional
```

The reason for this requirement is explained next.

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, which converts a `Datum` instance to its representation as a custom Java object, 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 (whether or not the class was created by JPublisher) 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 code for the `create()` method in the root class of a custom Java class hierarchy need not mention the subclasses, however. In fact, if it *did* mention the subclasses, 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 (equivalent to a type map) 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 runtime, the type is registered in the data structure. Because of this implicit mapping mechanism, no explicit type map, such as those required in `SQLData` scenarios, is required.

Important: 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 executed 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 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 for a subtype by JPublisher does not extend the reference classes of the base type.

Casting a Reference Type Instance into Another Reference Type

Revisiting the example in "ORAData Object Types and Inheritance" on page 3-19, `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 `cast()` method 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`. This is not the case, however, 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: 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, two problems would occur:

- Java would not permit the `getValue()` method in `StudentRef` to return a `Student` object when the method it would override 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.

It would not be sensible to 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

Because reference types do not follow the hierarchy of their related object types, there is a JPublisher limitation that you cannot convert directly from one reference type to another. For background information, this section explains how the generated `cast()` methods work to convert from one reference type to another.

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 code that could be used to convert from the reference type `XxxxRef` to the reference type `YyyyRef`:

```
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);
   ```

"Example: Manually Converting Between Reference Types", which immediately follows, provides sample code for such a conversion.

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 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 as discussed earlier, in "Manually Converting Between Reference Types" on page 3-22:

```
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 `SQLData` standard `readSQL()` and `writeSQL()` methods provide equivalent functionality to the `ORAData/ORADataFactory` `create()` and `toDatum()` methods for reading and writing data.

As is the case when JPublisher generates `ORAData` classes corresponding to a hierarchy of SQL object types, when JPublisher generates `SQLData` classes corresponding to a SQL hierarchy, the Java types follow the same hierarchy as the SQL types.

`SQLData` implementations do not, however, offer the implicit mapping intelligence that JPublisher automatically generates in `ORAData` classes (as described in "ORAData Object Types and Inheritance" on page 3-19).

In a `SQLData` scenario, you must manually provide a type map to ensure the proper 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 the *Oracle Database JDBC Developer's Guide and Reference* for information.

In addition, note that there is no support for strongly typed object references in a `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 on JPublisher-generated wrapper classes of using the SQL modifiers `FINAL`, `NOT FINAL`, or `NOT INSTANTIABLE`.

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 is so JPublisher users are able in all cases to customize generated Java wrapper classes by extending the classes and overriding the generated behavior.

Using the SQL modifier `NOT INSTANTIABLE` 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 instantiable SQL types.

4 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

For additional information about Oracle Database Web services, see *Oracle Database Java Developer's Guide*. For general information about Oracle features for Web services, see the *Oracle Application Server Web Services Developer's Guide*.

Summary of Support for Web Services Call-Ins to the Database

The following JPublisher features support Web services call-ins to code running in Oracle Database. They are described in detail elsewhere in this manual, as noted.

- Generation of Java interfaces

  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 API from which WSDL content is to be generated. Prior to Oracle Database 10*g*, JPublisher could generate classes but not interfaces.

  See "JPublisher Generation of Java Interfaces" on page 3-13.

- JPublisher styles and style files

  Style files, along with the related `-style` option, allow Java-to-Java type mappings to 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 "JPublisher Styles and Style Files" on page 2-22.

- 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 "Mapping of REF CURSOR Types and Result Sets" on page 2-7.

- 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. Using extended functionality of the `-sql` option, you can publish a specified 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 "Features to Filter JPublisher Output" on page 4-3.

- Support for calling Java classes in the database without PL/SQL call specs

  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, known as a call spec, that had to be created manually.

  See "Publishing Server-Side Java Classes" on page 1-16.

- Support for publishing SQL queries or DML statements

  JPublisher provides the `-sqlstatement` option to take a specified `SELECT`, `UPDATE`, `INSERT`, or `DELETE` statement and publish it as a method on a Java class that can be published as a Web service.

  See "Publishing SQL Queries or DML Statements" on page 1-16.

- Support for unique method names

  To meet Web services requirements, you can instruct JPublisher to disallow overloaded methods, instead always using unique method names.

  See "Generation of Package Classes and Wrapper Methods (-methods)" on page 5-27.

Support for Web Services Call-Outs from the Database

JPublisher supports Web services call-outs from Oracle Database. This is where Web services client code is written in SQL, PL/SQL, or Java to run inside the database, invoking Web services elsewhere. This support is through the `-proxywsdl` and `-httpproxy` options. In addition, the options `-proxyopts` and `-proxyclasses` are possibly relevant, but typically do not require any special settings for Web services so are not discussed here.

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.

> **Note:** See "Options to Facilitate Web Services Call-Outs" on page 5-34 for details about the options introduced here.

Features to Filter JPublisher Output

JPublisher supplies some options that allow you to filter what JPublisher produces, such as by 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 offer 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 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 "Declaration of Object Types and Packages to Translate (-sql)" on page 5-9 for additional information.

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 (no filtering), then list specific modes or types followed by minus signs ("-") for what to exclude. Alternatively, start with a "0" to include no possibilities by default (total filtering), then list specific modes or types followed by plus signs ("+") for what to allow. For example:

```
-filtertypes=1,.ORADATA-,.ORACLESQL-

-filtertypes=0,.CURSOR+,.INDEXBY+

-filtermodes=0,in+,return+

-filtermodes=1,out-,inout-
```

See "Method Filtering According to Parameter Modes (-filtermodes)" on page 5-24 and "Method Filtering According to Parameter Types (-filtertypes)" on page 5-24 for more information.

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 setting `-generatebean=true`, 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 "Code Generation Adherence to the JavaBeans Specification (-generatebean)" on page 5-25 for additional information.

Backward Compatibility and Migration

This section discusses issues of backward compatibility, compatibility between 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 created an application using an Oracle8*i* implementation of JPublisher, you probably will not be able to re-run JPublisher in Oracle Database 10*g* (or Oracle9*i*) and have the generated classes still work within your application.

In addition, to a lesser degree, there were changes in JPublisher functionality between Oracle9*i* and Oracle Database 10*g*. 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 Oracle9i and Oracle Database 10g
- Changes in JPublisher Behavior Between Oracle8i and Oracle9i
- JPublisher Backward Compatibility Modes and Settings

JPublisher Backward Compatibility

The JPublisher runtime 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 runtime.
- It can continue to compile against the current release of the JPublisher runtime.

If you use an earlier release of the JPublisher runtime and the Oracle JDBC drivers in generating code, then you can compile the code against that version of the JPublisher runtime. 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 Oracle9*i* and Oracle Database 10*g*

Regarding backward compatibility, a key difference in JPublisher behavior between Oracle9*i* and Oracle Database 10*g* is that now, by default, SQLJ source code is translated automatically, with `.sqlj` source files being invisible to the user. (See "JPublisher Usage of the Oracle SQLJ Implementation" on page 1-4.)

In addition, note the following changes in JPublisher behavior beginning in Oracle Database 10*g*:

- In Oracle9*i*, JPublisher generated SQLJ classes with a `protected` constructor with a boolean argument to specify whether the object must be initialized:

  ```
  protected BaseClass(boolean init) { ... }
  ```

  This constructor is removed in Oracle Database 10*g*, because it conflicts with constructor generation for a SQL object type with `BOOLEAN` attributes.
- Beginning in Oracle Database 10*g*, `SMALLINT` is mapped to `int` instead of to `short` in Java.

Changes in JPublisher Behavior Between Oracle8*i* and Oracle9*i*

Note the following JPublisher behaviors, beginning in Oracle9*i*:

- By default, JPublisher no longer declares the inner SQLJ connection context class `_Ctx` for every object type. Instead, it uses the connection context class `sqlj.runtime.ref.DefaultContext` 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 "More About Connection Contexts and Instances in SQLJ Classes" on page 3-9 for more information about the `getConnectionContext()` method.

  See "Changes in User-Written Subclasses of JPublisher-Generated Classes", immediately following, for additional information.
- Even with the setting `-methods=true`, non-SQLJ classes are generated if the underlying SQL object type or PL/SQL package does not define any methods. (But 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, be aware that several relevant changes were introduced, effective in Oracle9*i*, regarding how JPublisher generates code. You must make changes in any applications that have Oracle8*i* functionality if you want to use them in Oracle9*i* or Oracle Database 10*g*.

Note: If you use the `-compatible=8i` or `-compatible=both8i` option setting, you will not see the changes discussed here, and your application will continue to build and work as before. See "Backward Compatibility Option" on page 5-43 for information.

In general, however, it is advisable to make the transformation to the Oracle Database 10*g* JPublisher functionality, which insulates your user code from implementation details of JPublisher-generated classes.

Following are the changes to make in order to use your code in Oracle9*i* 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.

  Assume that in the example that follows, `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, instead 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. (See "Backward Compatibility Option" on page 5-43.) At the most elementary, this includes a setting to explicitly generate `.sqlj` files, which by default are transparent to users in Oracle Database 10*g*. 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

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 (as in Oracle8*i* and Oracle9*i* releases), you can use the JPublisher setting `-compatible=sqlj`.

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 "Option to Access SQLJ Functionality" on page 5-42.

Oracle9*i* Compatibility Mode

The JPublisher option setting `-compatible=9i` results in what is called *Oracle9i compatibility mode*. In this mode, JPublisher generates code that is compatible with Oracle9*i* SQLJ and JDBC releases. Additionally, in this mode, JPublisher typically produces `.sqlj` files that are visible to the user, as was the case with Oracle9*i* JPublisher.

JPublisher has the following functionality in 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 did in Oracle9*i*:

  ```
  protected BaseClass(boolean init) { ... }
  ```

  This constructor was removed in Oracle Database 10*g*, as described in "Changes in JPublisher Behavior Between Oracle9i and Oracle Database 10g" on page 4-5.
- The mapping in Java from `SMALLINT` reverts from `int` (the mapping in Oracle Database 10*g*) to `short`.

Oracle8*i* Compatibility Mode

Either the JPublisher setting `-compatible=both8i` or the setting `-compatible=8i` results in what is called *Oracle8i compatibility mode*. In this mode, JPublisher generates code that is compatible with Oracle8*i* SQLJ and JDBC releases. In addition, in this mode, JPublisher typically produces `.sqlj` files visible to the user, as was the case with Oracle8*i* JPublisher.

For use of this mode to be permissible, however, 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 executes under JDK 1.2 or higher and uses the SQLJ `runtime12.jar` library, or executes in the Oracle Database 10*g* release of the server-side Oracle JVM.
- You run JPublisher with the `-methods=false` or `-methods=none` setting.

Note the following functionality in 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 setting `-compatible=both8i`, 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 (as with `-methods=always`).
- JPublisher generates connection context declarations and connection context instances on every object wrapper class, as follows (as with `-context=generated`):

  ```
  #sql static context _Ctx;
  protected _Ctx _ctx;
  ```

- JPublisher provides a constructor in the wrapper class that takes a generic `ConnectionContext` instance (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 Oracle8*i* compatibility mode, it is advisable use the setting `-compatible=both8i`. 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 likely will be wrapped using `oracle.jdbc.Oracle`*`Xxxx`* interfaces. `CustomDatum` implementations do not support such wrapped connections.

Note: The setting `-compatible=both8i` requires a JDBC implementation from Oracle9*i* Release 1 (9.0.1) or higher.

Oracle8*i* compatibility mode is now the only way for a connection context instance `_ctx` 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 4–1 will produce many of the same results as using Oracle8*i* compatibility mode, which is described in the preceding section, "Oracle8i Compatibility Mode".

Table 4–1 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 (stored procedures).
-compatible=customdatum	For Oracle-specific wrapper classes, this setting results in JPublisher implementing the deprecated (but still supported) `oracle.sql.CustomDatum` and `CustomDatumFactory` interfaces instead of the `oracle.sql.ORAData` and `ORADataFactory` interfaces.
-dir=.	Setting this option to "." (a period, or "dot") results in generation of output files into a hierarchy under the current directory, as was the default behavior in Oracle8*i*.

Refer to the following for detailed descriptions of these options:

- "SQLJ Connection Context Classes (-context)" on page 5-15
- "Generation of Package Classes and Wrapper Methods (-methods)" on page 5-27
- "Backward-Compatible Oracle Mapping for User-Defined Types (-compatible)" on page 5-44
- "Output Directories for Generated Source and Class Files (-dir and -d)" on page 5-33

5 Command-Line Options and Input Files

This chapter describes the use and syntax details of JPublisher option settings and input files to specify program behavior, organized as follows:

- JPublisher Options
- Code Generation for Wrapper Class and PL/SQL Wrapper 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 Datatype Mappings
- Options for Type Maps
- 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

JPublisher Option Summary

Table 5–1 summarizes JPublisher options. For default values, the abbreviation "n/a" means "not applicable". The Category column refers to the corresponding conceptual area, indicating the section of this chapter where the option is discussed.

Table 5–1 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.	n/a	Type maps
-addtypemap	Appends an entry to the JPublisher user type map.	n/a	Type maps
-builtintypes	Specifies the datatype mappings (`jdbc` or `oracle`) for built-in datatypes that are non-numeric and non-LOB.	`jdbc`	Datatype mappings
-case	Specifies the case of Java identifiers that JPublisher generates.	`mixed`	Java code generation
-classpath	Adds to the Java classpath for JPublisher to use in resolving Java source and classes during translation and compilation.	Empty	Java environment
-compatible	Specifies a compatibility mode: explicit generation of `.sqlj` files, Oracle8*i* or Oracle9*i* compatibility mode, or the interface to implement (`ORAData` or `CustomDatum`) for Oracle mapping in generated classes. Modifies the behavior of `-usertypes=oracle`.	`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.	n/a	Java environment
-context	Specifies the class JPublisher uses for SQLJ connection contexts. This is either the `DefaultContext` class, a user-specified class, or a JPublisher-generated inner class.	`DefaultContext`	Connection
-defaulttypemap	Sets the default type map that JPublisher uses.	See "JPublisher User Type Map and Default Type Map" on page 2-5.	Type maps
-d	Specifies the root directory for placement of compiled class files.	Empty (all files directly in current directory)	Input/output
-dir	Specifies the root directory for placement of generated source files.	Empty (all files directly in current directory)	Input/output
-driver	Specifies the driver class that JPublisher uses for JDBC connections to the database.	`oracle.jdbc. OracleDriver`	Connection
-encoding	Specifies the Java encoding of JPublisher input files and output files.	The value of the system property `file.encoding`	Input/output
-endpoint	Specifies a Web service endpoint (used in conjunction with the `-proxywsdl` option).	n/a	Web services

Table 5–1 (Cont.) Summary of JPublisher Options

Option Name	Description	Default Value	Category
-filtermodes	Filters code generation according to specified parameter modes.	n/a	Java code generation
-filtertypes	Filters code generation according to specified parameter types.	n/a	Java code generation
-generatebean	Ensures that generated code conforms to the JavaBeans specification.	`false`	Java code generation
-genpattern	Defines naming patterns for generated code.	n/a	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 use in resolving the URL of a WSDL document, for access through a firewall (used in conjunction with the `-proxywsdl` option).	n/a	Web services
-input (or -i)	Specifies a file that lists the types and packages JPublisher translates.	n/a	Input files/items
-java	Specifies server-side Java classes for which JPublisher generates client-side classes.	n/a	Input files/items
-lobtypes	Specifies the datatype mappings (`jdbc` or `oracle`) that JPublisher uses for `BLOB` and `CLOB` types.	`oracle`	Datatype 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`	Datatype mappings
-methods	Determines whether JPublisher generates wrapper methods for stored procedures of translated SQL objects and PL/SQL packages. As secondary effects, 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 also settings to specify whether overloaded methods are allowed.	`all`	Java code generation
-numbertypes	Specifies the datatype mappings (`jdbc`, `objectjdbc`, `bigdecimal`, or `oracle`) that JPublisher uses for numeric datatypes.	`objectjdbc`	Datatype 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
-outarguments	Specifies "holder" type (arrays, JAX-RPC holders, or function returns) for Java implementation of PL/SQL output parameters.	`array`	Java code generation

Table 5–1 (Cont.) Summary of JPublisher Options

Option Name	Description	Default Value	Category
-package	Specifies the name of the Java package into which JPublisher generates Java wrapper classes.	n/a	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 specs, 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.	n/a	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).	n/a	Web services
-proxyopts	Used as input for the `-proxywsdl` and `-proxyclasses` options, to specify required layers of Java and PL/SQL wrappers and additional related settings.	`jaxrpc`	Web services
-proxywsdl	Specifies the URL of a WSDL document for which Web services client proxy classes and associated Java wrapper classes and PL/SQL wrappers are generated.	n/a	Web services
-serializable	Specifies whether 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.	n/a	Input files/items
-sqlj	Specifies SQLJ option settings for the JPublisher invocation of the SQLJ translator.	n/a	SQLJ
-sqlstatement	Specifies SQL queries or DML statements for which JPublisher generates Java classes, and optionally subclasses and interfaces, with appropriate methods.	n/a	Input files/items
-style	Specifies the name of a "style file" for Java-to-Java type mappings.	n/a	Datatype mappings

Table 5–1 (Cont.) Summary of JPublisher Options

Option Name	Description	Default Value	Category
-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.	n/a	Web services
-tostring	Specifies whether to generate a `toString()` method for object types.	`false`	Java code generation
-typemap	Specifies the JPublisher type map (a list of mappings).	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.	n/a	Input files/items
-url	Specifies the URL JPublisher uses to connect to the database.	`jdbc:oracle:oci:@`	Connection
-user (or -u)	Specifies an Oracle user name and password for connection.	n/a	Connection
-usertypes	Specifies the type mappings (`jdbc` or `oracle`) that JPublisher uses for user-defined SQL types.	`oracle`	Datatype mappings
-vm	Specifies a Java version, in case you want a version other than the default.	n/a	Java environment

JPublisher Option Tips

Be aware of the following usage notes for JPublisher options.

- JPublisher always requires the `-user` option (or `-u`, its shorthand equivalent).
- 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. This is *not* true for the following options, however, which are cumulative:

  `-sql` (or the deprecated `-types`)

  `-java`

  `-addtypemap` or `-adddefaulttypemap`

  `-style`
- In general, separate options and corresponding option values by an equals sign ("="). When the following options appear on the command line, however, you are also permitted to use a space as a separator:

  `-sql` (or `-s`), `-user` (or `-u`), `-props` (or `-p`), and `-input` (or `-i`)
- With the `-sqlj` option, however, you *must* use a space instead of an equals sign. (SQLJ settings following the `-sqlj` option use equals signs.) Following is an example; 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 your 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`.

  "Properties File Structure and Syntax" on page 5-51 describes the properties file.

- 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 follows these 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.
- 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 here are entered as shown. These include "`.`" and "`@`", for example.

Options for Input Files and Items to Publish

This section documents JPublisher options that specify key input—either JPublisher input files (`INPUT` files or properties files) or items to publish (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` (deprecated)

These options are discussed in alphabetical order.

File Containing Names of Objects and Packages to Translate (-input)

```
-input=filename
-i filename
```

Both formats are synonymous. The second one is provided for convenience as a command-line abbreviation.

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, listing the types that need Java class definitions.

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. For more information on this topic, see "Translating Additional Types" on page 5-56.

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.

For more information about the syntax of the `INPUT` file, see "INPUT File Structure and Syntax" on page 5-53.

Declaration of Server-Side Java Classes to Translate (-java)

`-java=`*`class_or_package_list`*

As described in "Publishing Server-Side Java Classes" on page 1-16, you can use the `-java` option to create client-side stub classes to use in accessing 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 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, 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, as with the `-sql` option, you must also specify `-user` and `-url` settings for a database connection.
- Functionality of the `-java` option requires the library `sqljutl.jar` to be loaded in the database. (See "Required Packages and JAR Files in the Database" on page 1-7.)
- It is advisable to use the same JDK on the client as in 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`, 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 likely the most common scenario.

As an example, assume that you want to call the following method in 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 method `Class.forName()` to search for the class among system classes in the JVM (typically 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 package `oracle.jpub.reflect`, 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[]` and `String[]`, for example) 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 all be part of the function result.

– Exceptions are properly returned.

– The method invocation uses invoker's rights. (There is no tuning to obtain definer's rights.) See the *Oracle Database Java Developer's Guide* for information about invoker's rights and definer's rights.

Input Properties File (-props)

```
-props=filename
-p filename
```

Both formats are synonymous. The second one is provided for convenience as a command-line abbreviation.

The `-props` option, entered on the command line, 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 at the point of the `-props` option.

If more than one properties file appears on the command line, JPublisher processes them with the other command-line options, in the order in which they appear.

For information on the contents of the properties file, see "Properties File Structure and Syntax" on page 5-51.

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 encoding `8859_1`. This is a feature of Java in general, not JPublisher in particular. You can, however, use Unicode escape sequences in a properties file.

Declaration of Object Types and Packages to Translate (-sql)

```
-sql={toplevel|object_type_and_package_translation_syntax}
-s {toplevel|object_type_and_package_translation_syntax}
```

Use the `-sql` option to specify SQL user-defined types (objects or collections) or PL/SQL packages to publish, optionally specifying user subclasses or interfaces to generate. You can publish all or a specified subset of a PL/SQL package. The two formats of this option (`-sql` and `-s`) are synonymous. The `-s` format is provided for convenience as a 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. More information on the `toplevel` keyword is provided later in this section.

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.

In this section, the `-sql` option is explained in terms of the equivalent `INPUT` file syntax. "Understanding the Translation Statement" on page 5-53 discusses INPUT file syntax.

You can use the any of the following syntax modes:

- `-sql=`*`name_a`*

  Or, in an `INPUT` file, use:

  ```
  SQL name_a
  ```

  JPublisher publishes *`name_a`*, naming the generated class according to default settings.

- `-sql=`*`name_a:class_c`*

  Or, in an `INPUT` file, use:

  ```
  SQL name_a AS class_c
  ```

  JPublisher publishes *`name_a`* as the generated Java class *`class_c`*.

- `-sql=`*`name_a`*`:`*`class_b`*`:`*`class_c`*

  Or, in an `INPUT` file, use:

  ```
  SQL name_a GENERATE class_b AS class_c
  ```

  In this case, *`name_a`* must represent an object type. JPublisher generates the Java class *`class_b`* and a stub *`class_c`* that extends *`class_b`*. You provide the code for *`class_c`*, which is used to represent *`name_a`* in your Java code.

- `-sql=`*`name_a`*`:`*`class_b`*`#`*`intfc_b`*

- `-sql=`*`name_a`*`:`*`class_b`*`:`*`class_c`*`#`*`intfc_c`*

  Use either of these syntax formats to have JPublisher generate a Java interface. This feature is particularly useful for Web services. See "JPublisher Generation of Java Interfaces" on page 3-13.

  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`*.

  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 non-case-sensitive SQL names are supported on the JPublisher command line. If a user-defined type was defined in a case-sensitive way (in quotes) in SQL, then you must specify the name in the JPublisher `INPUT` file instead of on the command line, and in quotes. See "INPUT File Structure and Syntax" on page 5-53 for information.
- If your desired class and interface names follow a pattern, you can use the `-genpattern` command-line option for convenience. See "Class and Interface Naming Pattern (-genpattern)" on page 5-25.

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 command this as follows:

```
SQL CORPORATION
SQL EMPLOYEE AS OracleEmployee
SQL ADDRESS GENERATE JAddress AS MyAddress
```

And JPublisher executes the following:

- It creates a wrapper class for the `CORPORATION` package.
- It translates the object type `EMPLOYEE` as `OracleEmployee`.
- It generates an object reference class `OracleEmployeeRef`.
- It translates `ADDRESS` as `JAddress`, but generates code and references so that `ADDRESS` objects will be represented by the `MyAddress` class.
- It generates a `MyAddress` stub, where you will write your custom code, that extends `JAddress`.
- It 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 `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, enter the following command.

```
-sql=schema_name.toplevel
```

In this example, `schema_name` is the name of the schema containing the top-level subprograms.

There are also 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 plus signs ("+") between stored procedure names, as shown.

Alternatively, for the SQL top level, use:

```
-sql=toplevel(proc1+proc2+proc3+...)
```

Following is the syntax for a JPublisher `INPUT` file. Use commas between stored procedure names, as shown:

```
SQL plsql_package (proc1, proc2, proc3, ...) AS ...
```

Notes:

- In an `INPUT` file, put a stored procedure name in quotes (for example, "*proc1*") if it is case-sensitive. JPublisher assumes that names 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 any stored procedure whose name starts with "`myfunc`", such as `myfunc1`.

You can also specify the subset according to stored procedure names and argument types, using the following syntax:

```
myfunc(sqltype1, sqltype2, ...)
```

In this case, only stored procedures that match in name, as well as in the number and types of arguments, will be published. For example:

```
-sql=mypackage(myfunc1(NUMBER, CHAR)+myfunc2(VARCHAR2))
```

Declaration of SQL Statements to Translate (-sqlstatement)

```
-sqlstatement.class=ClassName:UserClassName#UserInterfaceName
-sqlstatement.methodName=sqlStatement
-sqlstatement.return={both|resultset|beans}
```

The JPublisher `-sqlstatement` option enables you to publish `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements as Java methods. JPublisher generates SQLJ classes for this functionality.

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, or a user interface for the generated class (or subclass, if applicable) to implement, 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 methods. "Generic" means the column types of the result set are unknown or unspecified. For queries, however, the column types are actually known, allowing 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. See "Class and Interface Naming Pattern (-genpattern)" on page 5-25.

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. See "JPublisher Styles and Style Files" on page 2-22. If a SQL statement uses Oracle datatype `X`, which corresponds to Java type `Y`, and type `Y` is mapped to Java type `Z` in the style file, then methods generated as a result of the `-sqlstatement` option will use Java type `Z`, not type `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 "Mapping of REF CURSOR Types and Result Sets" on page 2-7.

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 (wraparound) JPublisher command to publish an `INSERT` statement:

```
% jpub  -u scott/tiger -style=webservices10
        -sqlstatement.addEle="insert into xmltab values(:{a sys.xmltype})"
```

This command directs 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 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, you can set the `-vm` and `-compiler-executable` options to specify a JDK 1.4 JVM and compiler. See "Java Environment Options" on page 5-45.

Declaration of Object Types to Translate (-types)

`-types=`*`type_translation_syntax`*

Note: The `-types` option is currently supported for compatibility, but deprecated. Use the `-sql` option instead.

You can use the `-types` option, *for SQL object types only*, when you do not need the generality of an `INPUT` file. The `-types` option lets you list one or more individual object types that you want JPublisher to translate. 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 "Understanding the Translation Statement" on page 5-53.

Enter `-types=...` on the command line, followed by one or more object type translations 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 relating to the database connection that JPublisher uses: `-context`, `-driver`, `-url`, and `-user`.

These options are discussed in alphabetical order.

SQLJ Connection Context Classes (-context)

`-context={generated|`**`DefaultContext`**`|`*`user_defined`*`}`

The `-context` option specifies the connection context class that JPublisher uses, and possibly declares, for SQLJ classes that JPublisher produces.

The setting `-context=DefaultContext` 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 that 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. See "More About Connection Contexts and Instances in SQLJ Classes" on page 3-9 for information about these methods.

The setting `-context=generated` results in an inner class declaration for the connection context class `_Ctx` in all SQLJ classes generated by JPublisher. So, each class uses its own SQLJ connection context class. (Also see "More About Connection Contexts and Instances in SQLJ Classes" on page 3-9.) This setting may be appropriate for Oracle8*i* compatibility mode (see the notes immediately following), but is otherwise not recommended. Using the `DefaultContext` class or a user-defined class avoids having additional connection context classes generated.

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 (through a `-compatible` setting of `sqlj`, `9i`, `8i`, or `both8i`) and, therefore, use `.sqlj` files and the SQLJ translator directly, 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 or under JDK 1.2 or higher. This is *not* the case with the setting `-context=generated`, 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 higher 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.

JDBC Driver Class for Database Connection (-driver)

```
-driver=driver_class_name
```

The `-driver` option specifies the driver class that JPublisher uses for JDBC connections to the database. The default is:

```
-driver=oracle.jdbc.OracleDriver
```

This setting is appropriate for any Oracle JDBC driver.

Connection URL for Target Database (-url)

```
-url=URL
```

You can use the `-url` option to specify the URL of the database to which you want to connect. The default value is:

```
-url=jdbc:oracle:oci:@
```

To specify the 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. (The use of SIDs is deprecated in Oracle Database 10*g*, but is still supported for backward compatibility. Their use is of the form *`host`*`:`*`port`*`:`*`sid`*.)

Note: Use "`oci`" in the connect string for the Oracle JDBC OCI driver in any new code. For backward compatibility, however, "`oci8`" is still accepted for Oracle8*i* drivers.

User Name and Password for Database Connection (-user)

```
-user=username/password
-u username/password
```

Both formats are equivalent. The second one is provided for convenience as a command-line shortcut.

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, JPublisher prints an error message and stops execution.

For example, the following command line directs JPublisher to connect to your database with user name `scott` and password `tiger`:

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

Options for Datatype Mappings

The following options control which datatype mappings 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 (possibly 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 datatype mappings for numeric types.
- The `-lobtypes` option controls datatype mappings for the `BLOB`, `CLOB`, and `BFILE` types.
- The `-builtintypes` option controls datatype mappings for non-numeric, non-LOB, predefined SQL and PL/SQL types.

These four options are known as the type-mapping options, and are discussed in alphabetical order in the subsections that follow. (Another, less flexible option, `-mapping`, is also discussed. It is deprecated, but still supported for compatibility with older releases of JPublisher.)

For an object type, JPublisher applies the mappings specified by the type mapping options to the object attributes and to the arguments and results of any methods (stored procedures) included with the object. The mappings control the types that the generated accessor methods support; that is, what 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 not supported by Web services, in the JPublisher-generated base class, then that Java type is mapped to a Java type that *is* supported by Web services, in the JPublisher-generated user subclass. See "JPublisher Styles and Style Files" on page 2-22 for related information.

Mappings For Built-In Types (-builtintypes)

`-builtintypes={`**`jdbc`**`|oracle}`

The `-builtintypes` option controls datatype mappings for all the built-in datatypes except the LOB types (controlled by the `-lobtypes` option) and the different numeric types (controlled by the `-numbertypes` option). Table 5–2 lists the datatypes affected by the `-builtintypes` option and shows their Java type mappings for `-builtintypes=oracle` and `-builtintypes=jdbc` (the default).

Table 5–2 Mappings for Types Affected by the -builtintypes Option

SQL Datatype	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 5–2 (Cont.) Mappings for Types Affected by the -builtintypes Option

SQL Datatype	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 (-lobtypes)

`-lobtypes={jdbc|`**`oracle`**`}`

The `-lobtypes` option controls datatype mappings for LOB ("large object") types. Table 5–3 shows how these types are mapped for `-lobtypes=oracle` (the default) and for `-lobtypes=jdbc`.

Table 5–3 Mappings for Types Affected by the -lobtypes Option

SQL Datatype	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 (-numbertypes)

`-numbertypes={jdbc|`**`objectjdbc`**`|bigdecimal|oracle}`

The `-numbertypes` option controls datatype mappings for numeric SQL and PL/SQL types. Four choices are available:

- In JDBC mapping, most numeric datatypes 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 (the default), most numeric datatypes 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`.
- In `BigDecimal` mapping, all numeric datatypes are mapped to `java.math.BigDecimal`.
- In Oracle mapping, all numeric datatypes are mapped to `oracle.sql.NUMBER`.

Table 5–4 lists the datatypes affected by the `-numbertypes` option, and shows their Java type mappings for `-numbertypes=jdbc` and `-numbertypes=objectjdbc` (the default).

Table 5–4 Mappings for Types Affected by the -numbertypes Option

SQL Datatype	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 (-usertypes)

`-usertypes={`**`oracle`**`|jdbc}`

The `-usertypes` option controls whether JPublisher implements the Oracle `ORAData` interface or the standard `SQLData` interface in generated classes for user-defined types.

When `-usertypes=oracle` (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 higher, 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`. See "Backward-Compatible Oracle Mapping for User-Defined Types (-compatible)" on page 5-44.

Mappings for All Types (-mapping)

`-mapping={jdbc|`**`objectjdbc`**`|bigdecimal|oracle}`

Note: This option is deprecated in favor of the more specific type mapping options: `-usertypes`, `-numbertypes`, `-builtintypes`, and `-lobtypes`. It is still supported, however, for backward compatibility.

The `-mapping` option specifies mapping for all datatypes, so offers little flexibility between types.

The setting `-mapping=oracle` 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, as summarized in Table 5–5.

Table 5–5 Relation of -mapping Settings to Settings of Other Mapping Options

-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 (-style)

```
-style=stylename
```

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.

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.

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 "JPublisher Styles and Style Files" on page 2-22 for more information.

Options for Type Maps

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
```

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.

Related options, discussed in alphabetical order in subsections that follow, 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.

For more information about the first format shown, for mapping of an OPAQUE type, see "Type Mapping Support for OPAQUE Types" on page 2-11. The second and third formats, using *`max_length`* and *`elem_size`* for scalar indexed-by tables, are discussed in "Type Mapping Support for Scalar Indexed-by Tables Using JDBC OCI" on page 2-13. The last format, for mapping of a PL/SQL type unsupported by JDBC, is explained in "Type Mapping Support Through PL/SQL Conversion Functions" on page 2-15. In the type map syntax, *`sql_to_plsql_func`* and *`plsql_to_sql_func`* are for functions that convert between SQL and PL/SQL.

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. See "JPublisher User Type Map and Default Type Map" on page 2-5 for additional information about these type maps and how JPublisher uses them.

Additional Entry to the Default Type Map (-adddefaulttypemap)

`-adddefaulttypemap=`*`list_of_typemap_entries`*

Use this 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 format for type map entries is described in the top-level section, "Options for Type Maps" on page 5-20.

Additional Entry to the User Type Map (-addtypemap)

`-addtypemap=`*`list_of_typemap_entries`*

Use this option to append an entry or a comma-delimited list of entries to the JPublisher user type map. The format for type map entries is described in the top-level section, "Options for Type Maps" on page 5-20.

Default Type Map for JPublisher (-defaulttypemap)

`-defaulttypemap=`*`list_of_typemap_entries`*

JPublisher uses this option internally to set up predefined type map entries in the default type map.

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 "JPublisher User Type Map and Default Type Map" on page 2-5 for additional information, including a caution about conflicts between the type maps.

Replacement of the JPublisher Type Map (-typemap)

-typemap=*list_of_typemap_entries*

Use this option to specify an entry or a comma-delimited list of entries to set up the user type map.

The difference between the `-typemap` option and the `-addtypemap` option is that `-typemap` replaces the existing user type map with the specified entries; `-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. See "JPublisher User Type Map and Default Type Map" on page 2-5 for additional information, including a caution about conflicts between the type maps.

The format for type map entries is described in the top-level section, "Options for Type Maps" on page 5-20.

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 (-access)

-access={**public**|protected|package}

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.

JPublisher uses the possible option settings as follows:

- `public` (default): Methods are generated with the `public` access modifier.
- `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. Perhaps you are providing your own 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, VARRAYs, and nested tables are not affected by the value of the `-access` option.

Case of Java Identifiers (-case)

`-case={`**`mixed`**`|same|lower|upper}`

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.

Table 5–6 describes the possible values for the `-case` option.

Table 5–6 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 lower case. An underscore (_), dollar sign ($), or any character illegal in Java constitutes a word-unit boundary and is silently removed. 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, overriding the `-case` option.

Method Filtering According to Parameter Modes (-filtermodes)

`-filtermodes=`*`list_of_modes_to_filter_out_or_filter_in`*

In some cases, particularly for code generation 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. (Also see the `-filtertypes` option, following.)

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 (no filtering), then list specific modes or types followed by minus signs ("-") for what to exclude. Alternatively, start with a "0" to include no possibilities by default (total filtering), then list specific modes or types followed by plus signs ("+") for what to allow.

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 (-filtertypes)

`-filtertypes=`*`list_of_types_to_filter_out_or_filter_in`*

In some cases, particularly for code generation 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. (Also see the `-filtermodes` option, preceding.)

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 (no filtering), then list specific modes or types followed by minus signs ("-") for what to exclude. Alternatively, start with a "0" to include no possibilities by default (total filtering), then list specific modes or types followed by plus signs ("+") for what to allow.

In the following examples, the first filters out only `.ORADATA` and `.ORACLESQL`. The second 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, so 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 (-generatebean)

-generatebean={true|**false**}

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 setting `-generatebean=true`, 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 types, or indexed-by table types, 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;
```

Method names are changed 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 (-genpattern)

-genpattern=*pattern_specifications*

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.

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
```

"`%1`", by definition, refers to the default base names that JPublisher would create for each SQL type. By default, JPublisher would create the Java type `Person` for the SQL type `PERSON`, the Java type `Student` for the SQL type `STUDENT`, and the Java type `GradStudent` for the SQL type `GRAD_STUDENT`. So "`%1Base`" becomes `PersonBase`, `StudentBase`, and `GradStudentBase`, respectively. Similarly 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.

Following is an example that 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 (-gensubclass)

-gensubclass={**true**|false|force|call-super}

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.

For `-gensubclass=true` (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, 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; JPublisher implementation details should 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 (-methods)

`-methods={`**`all`**`|none|named|always,`**`overload`**`|unique}`

The settings of the `-methods` option determine two things regarding the generation of wrapper methods:

- Whether JPublisher generates wrapper methods for methods (stored procedures) in SQL object types and PL/SQL packages (through a setting of `all`, `none`, `named`, or `always`)
- Whether overloaded method names are allowed (through a setting of `overload` or `unique`)

For `-methods=all` (the default among the first group of settings), JPublisher generates wrapper methods for all the methods in the 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, but a non-SQLJ class if not. (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 to 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 setting `-methods=unique` specifies that all method names must be unique. This is required for Web services, for example.

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, they are published as follows, using a method renaming mechanism based on the first letter of the return type and argument types, as "Translation of Overloaded Methods" on page 3-5 describes.

```
String foo(String a);
java.math.BigDecimal fooBBB(java.math.BigDecimal x, java.math.BigDecimal y);
```

To specify a setting of `all`, `none`, `named`, or `always` at the same time as you specify a setting of `overload` or `unique`, use a comma to separate the two settings, as in the following example:

```
-methods=always,unique
```

You can specify the `-methods` option on the command line or in a properties file.

Omission of Schema Name from Name References (-omit_schema_names)

```
-omit_schema_names
```

In publishing SQL user-defined (object and collection) types, 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. In this case, 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.

or:

- You declare the user-defined SQL type with a schema name on the command line or `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 was 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 executes, 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, the type is looked up in the schema associated with the current connection when the code generated by JPublisher is executed. If the SQL type name does include the schema name, 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 runtime 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 will 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, or do nothing to leave it disabled.

Holder Types for Output Arguments (-outarguments)

`-outarguments={`**`array`**`|holder|return}`

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 (the default)
- JAX-RPC holder types
- Function returns

The `-outarguments` option enables you to specify which mechanism to use, through a setting of `array`, `holder`, or `return`, respectively. This feature is particularly useful for Web services.

See "JPublisher Treatment of Output Parameters" on page 3-1 for details about the meaning and ramifications of each of these settings.

Name for Generated Java Package (-package)

`-package=`*`package_name`*

The `-package` option specifies the name of the Java package that JPublisher generates. The name appears in a package declaration in each generated class.

If you use the `-dir` and `-d` options, the directory structure in which JPublisher places generated files reflects the package name as well as the `-dir` and `-d` settings. See "Output Directories for Generated Source and Class Files (-dir and -d)" on page 5-33 for information about those options.

> **Notes:**
>
> - 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.
> - 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.

Example 1 Assume the following command line:

```
% 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`.

Additionally, the `Person` class includes the following package declaration:

```
package c.d;
```

Example 2 Now assume the following command line:

```
% jpub -dir=/a/b -d=/a/b -package=c.d -sql=PERSON:Person -input=myinputfile
```

And assume `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 supply a package name, then JPublisher does not generate any package declaration. Output `.java` files are placed directly into the directory specified by the `-dir` option (or into the current directory by default), and 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. (It may be an attribute of the requested type, for example.) 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 in 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 (-serializable)

`-serializable={true|`**`false`**`}`

The `-serializable` flag specifies whether the Java classes that JPublisher generates for SQL object types implement the `java.io.Serializable` interface. The default setting is `-serializable=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 severed, 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 (-tostring)

`-tostring={true|`**`false`**`}`

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`.

PL/SQL Code Generation Options

This section documents the following options that specify JPublisher behavior in generating PL/SQL code. These options are mostly to support Java calls to stored procedures that use PL/SQL types. They 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, to allow access by JDBC.

- `-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.

File Names for PL/SQL Scripts (-plsqlfile)

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

This option specifies the name of a wrapper script and a dropper script generated by JPublisher. 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 (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, they are overwritten. If no file names are specified, JPublisher writes to files named `plsql_wrapper.sql` and `plsql_dropper.sql`.

JPublisher outputs 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 (-plsqlmap)

`-plsqlmap={`**`true`**`|false|always}`

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 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 setting can be any of the following:

- `true` (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 if PL/SQL types are used only as `IN` parameters or for the function return, then generated code instead calls the stored procedure directly, processing its 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 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 inaccessible from Java in the original package.

See "Type Mapping Support Through PL/SQL Conversion Functions" on page 2-15 and "Direct Use of PL/SQL Conversion Functions Versus Use of Wrapper Functions" on page 2-20 for related information.

Package for Generated PL/SQL Code (-plsqlpackage)

`-plsqlpackage=`*`name_of_PLSQL_package`*

The `-plsqlpackage` option specifies the name of a PL/SQL package into which JPublisher places any generated PL/SQL code, including PL/SQL wrappers (call specs), conversion functions to convert between PL/SQL types and SQL types, wrapper functions to wrap stored procedures that use PL/SQL types, and table functions.

(Regarding conversion functions and wrapper functions, see "Type Mapping Support Through PL/SQL Conversion Functions" on page 2-15 and "Direct Use of PL/SQL Conversion Functions Versus Use of Wrapper Functions" on page 2-20 for related information.)

By default, JPublisher uses the package `JPUB_PLSQL_WRAPPER`.

Note that it is your responsibility to create this package in the database by running the SQL script generated by JPublisher. See "File Names for PL/SQL Scripts (-plsqlfile)" on page 5-31.

Input/Output Options

This section documents options relating to JPublisher input and output files and locations, listed in the order in which they are discussed:

- The `-compile` option, if you want to suppress compilation (and optionally SQLJ translation as well, if JPublisher is in a backward compatibility mode)

- The `-dir` option, to specify where generated source files are placed
- The `-d` option, to specify where compiled class files are placed
- The `-encoding` option, to specify the Java character encoding of the `INPUT` file JPublisher reads and the `.sqlj` and `.java` files JPublisher writes

No Compilation or Translation (-compile)

`-compile={`**`true`**`|false|notranslate}`

Use this option to suppress the compilation of generated `.java` files and, for backward compatibility modes, to optionally suppress the translation of generated `.sqlj` files. With the default `true` setting, all generated classes are compiled into `.class` files.

If you are in a backward compatibility mode (`-compatible=both8i`, `8i`, `9i`, or `sqlj`), then you can use a setting of `-compile=notranslate` 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, using either the JPublisher `-sqlj` option or the SQLJ command-line utility directly. Or you can use a setting of `-compile=false` 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` setting (`oradata`), you can use a setting of `-compile=false` to skip compilation. In this scenario, the `notranslate` setting is not supported, given that visible `.sqlj` files are not produced if you are not in a backward compatibility mode.

See "Backward Compatibility Option" on page 5-43 for information about the `-compatible` option, and "Option to Access SQLJ Functionality" on page 5-42 for information about the `-sqlj` option.

Output Directories for Generated Source and Class Files (-dir and -d)

`-dir=`*`directory_path`*
`-d=`*`directory_path`*

Use the `-dir` option to specify the root of the directory tree within which JPublisher places `.java` source files (or `.sqlj` source files for backward compatibility modes). A setting of "." (a period, or "dot") 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 "." setting.

For each option, with any nonempty setting, JPublisher also uses package information from the `-package` option or any package name included in a `SQL` option setting in the `INPUT` file to determine the complete directory hierarchy for generated files. Also see "Name for Generated Java Package (-package)" on page 5-29.

For example, consider the following JPublisher (wraparound) command line:

```
% 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). Or you can 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 (-encoding)

`-encoding=`*`name_of_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 system property `file.encoding` or, if this property is not set, `8859_1` (ISO Latin-1).

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. Under 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 encoding `8859_1`. This is a feature of Java in general, not JPublisher in particular. You can, however, 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, this might be to access Web services client code from inside the database, referred to as *Web services call-outs*. Details are covered in the following subsections.

- Mechanisms Used in Exposing Java to PL/SQL
- Classes for Java and PL/SQL Wrapper Generation (-proxyclasses)
- Settings for Java and PL/SQL Wrapper Generation (-proxyopts)
- WSDL Document for Java and PL/SQL Wrapper Generation (-proxywsdl)
- Web Services Endpoint (-endpoint)
- Proxy URL for WSDL (-httpproxy)
- Superuser for Permissions to Run Client Proxies (-sysuser)

Here is a summary of the relevant options and how they relate to each other:

- `-proxyclasses=`*`class1,class2,...,classN`*

  This option specifies Java classes for which Java (as necessary) 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, any time 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 use the `-proxyclasses` option directly, 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 for steps 2 and 3, 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 protocol.

Notes:

- The features described here require the library `utl_dbws_jserver.jar` to be installed in Oracle Database 10g. See "Required Packages and JAR Files in the Database" on page 1-7 for additional information.
- 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.
- Also see "Code Generation for Wrapper Class and PL/SQL Wrapper Options" on page 5-46. And for related code generation examples, see "Generated Code: Java and PL/SQL Wrappers for Web Services" on page A-10 and "Generated Code: Java and PL/SQL Wrappers for General Use" on page A-22.

Mechanisms Used in Exposing Java to PL/SQL

Part of the JPublisher functionality for Web services call-outs from Oracle Database is to expose Java functionality to PL/SQL so that a Web services client can be called from PL/SQL. Or there may be other reasons for wanting to access Java from PL/SQL stored procedures.

JPublisher implements this functionality by generating PL/SQL wrappers, otherwise known as PL/SQL call specs. A *PL/SQL wrapper* is a PL/SQL package that can invoke methods of one or more given Java classes. (See the *Oracle Database Java Developer's Guide* for additional information.)

PL/SQL supports only static methods. Java classes with only static methods (or for which you want to expose only static methods) can be wrapped in a straightforward manner. For Java classes that have instance methods that you want to expose, however, 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 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 for 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.

  See "Wrapper Class Generation without Handles" on page 5-46 for details about code generation.
- Instances of the wrapped class can be identified through *handles* (ID numbers). JPublisher uses `long` numbers as handles, and creates static methods in the

wrapper class with modified method signatures (in comparison to the signatures of the original instance methods) to also take 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.

See "Wrapper Class Generation with Handles" on page 5-47 for a details about code generation.

Classes for Java and PL/SQL Wrapper Generation (-proxyclasses)

`-proxyclasses[@`*`server`*`]=`*`class_or_jar_list`*

You can use this option to specify a comma-delimited list of Java classes (either loose classes or JAR files) for which JPublisher creates PL/SQL wrappers. Depending on the situation, JPublisher may 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.

Note: When the `-proxywsdl` option is used to create and wrap Web services client proxy classes, JPublisher supplies the list of proxy classes to the `-proxyclasses` option behind the scenes. Explicit use of the `-proxyclasses` option is for more general use, not necessarily to support Web services.

To summarize: for each class being processed, the following are generated, depending on settings of the `-proxyopts` option.

- A PL/SQL wrapper to allow access from PL/SQL (always produced)
- 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 method calling sequences.

For Web services, use `-proxywsdl` instead of `-proxyclasses`, which is used behind the scenes and is set automatically as appropriate. Typically, the default `-proxyopts` setting is sufficient in this scenario.

Alternatively, you can use `-proxyclasses` directly, for general purposes, any time you want to create PL/SQL wrappers for Java classes. In this scenario, set `-proxyclasses` and `-proxyopts` as appropriate.

See "Mechanisms Used in Exposing Java to PL/SQL" on page 5-36 for related information.

In using the `-proxyclasses` option directly, you can specify JAR files, client-side Java classes, server-side Java classes, or server-side Java packages. The option has two forms, as follows.

- `-proxyclasses=`*`class_or_jar_list`*, a comma-delimited list of classes or JAR files
- `-proxyclasses@`*`server`*`=`*`class_list`*, a comma-delimited list of server-side classes at the specified server

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.*)`

See "Generated Code: Java and PL/SQL Wrappers for General Use" on page A-22 for code examples.

Settings for Java and PL/SQL Wrapper Generation (-proxyopts)

`-proxyopts=`*`setting1,setting2,...`*

This option is used as input by the `-proxywsdl` and `-proxyclasses` options and specifies JPublisher behavior in generating wrapper classes and PL/SQL wrappers for server-side Java classes. This is usually, but not necessarily, for Web services.

The `-proxyopts` option uses the basic settings listed immediately below, which can be used individually or in combinations. In this discussion, "processed classes" are the classes being wrapped—either Web services client proxy classes according to a WSDL document, if you use the `-proxywsdl` option, or the classes that you directly specify through the `-proxyclasses` option.

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.

- Use the `static` setting to specify treatment of static methods of processed classes. This functions as follows:

  `static`: 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. (This is not appropriate for Web services.)

  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`: With this setting, only a single default instance of each wrapped class is used during runtime. Therefore, 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.

Without one of these settings (or a `jaxrpc` or `soap` setting, which implies `single`), instance methods are ignored. 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.

See "Mechanisms Used in Exposing Java to PL/SQL" on page 5-36 for information about the handle and singleton mechanisms.

- Use the `jaxrpc` or `soap` setting to publish instance methods of Web services client proxy classes. These settings function as follows:

  `jaxrpc` (default setting): This is a convenience setting for wrapping JAX-RPC client proxy classes, which is appropriate for use with 10.0.x releases of Oracle Application Server 10*g*. JPublisher creates a Java wrapper class for each processed class, as well as creating the PL/SQL wrapper. Client proxy classes do not have static methods to be published, and, by default, instance methods are published using the singleton mechanism. So, when processing JAX-RPC client proxy classes, `-proxyopts=jaxprc` 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 for wrapping SOAP client proxy classes instead of JAX-RPC client proxy classes. This is appropriate for use with the 9.0.4 release of Oracle Application Server 10g, or any earlier releases of the application server.

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, maintaining instance methods as 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 more typical to explicitly use the `-proxyopts` option with the `-proxyclasses` option than it is to explicitly use `-proxyopts` with the `-proxywsdl` option. For use of `-proxywsdl` with 10.0.x releases of Oracle Application Server 10*g*, the default `-proxyopts=jaxrpc` setting is sufficient.

There are additional, more advanced, `-proxyopts` settings as well:

- `noload`: Do not load generated code into the database. (By default it is loaded.)
- `recursive`: When processing a class that extends another class, also create wrappers (PL/SQL, and Java if appropriate) for inherited methods.
- `tabfun`: Use this with the `jaxrpc` or `soap` setting to have JPublisher 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. Also see "Code Generation for Table Functions" on page 5-50.
- `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. A function declared as deterministic returns the same result given the same inputs, by definition, so that a subsequent call with the same parameter settings can be bypassed, reusing a cached result from a previous call instead.
- `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 you want to support. You can use commas or hyphens, 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.

Here is an example using the `jaxrpc` basic setting by default. It also uses table functions and indicates that wrapped methods are deterministic:

```
-proxyopts=tabfun,deterministic
```

Here is an example that explicitly sets `static` mode (presumably processing classes that are not client proxy classes) and specifies that generated code is not loaded into the database:

```
-proxyopts=static,noload
```

WSDL Document for Java and PL/SQL Wrapper Generation (-proxywsdl)

-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 accordingly, uses the `-proxyopts` setting (often just the default 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 runtime to access the Web services specified in the WSDL document.
2. As appropriate or necessary, creates Java wrapper classes for the Web services client proxy classes. For each proxy class that has instance methods (as is typical), 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 (call specs) for the generated classes, to make them accessible from PL/SQL. Because PL/SQL supports only static methods, this step requires the wrapping of instance methods by static methods that 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.

Notes: 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 of the application server, set `-proxyopts=soap`.

The `-endpoint` option is typically used in conjunction with the `-proxywsdl` option. See the next section, "Web Services Endpoint (-endpoint)".

Refer to the following sections for additional related information:

- "Mechanisms Used in Exposing Java to PL/SQL" on page 5-36
- "Classes for Java and PL/SQL Wrapper Generation (-proxyclasses)" on page 5-37
- "Settings for Java and PL/SQL Wrapper Generation (-proxyopts)" on page 5-38

Also see "Generated Code: Java and PL/SQL Wrappers for Web Services" on page A-10 for code examples.

Web Services Endpoint (-endpoint)

```
-endpoint=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.

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 statement below
// ((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.

This endpoint example is taken from the wrapper code in "Generated Code: Java and PL/SQL Wrappers for Web Services" on page A-10.

Proxy URL for WSDL (-httpproxy)

`-httpproxy=`*`proxy_URL`*

If a WSDL document used for Web services call-outs is accessed through a firewall, use this option in conjunction with the `-proxywsdl` option to specify a proxy URL to use in resolving the URL of the WSDL document. For example:

```
% jpub ... -httpproxy=http://www-proxy.oracle.com:80
```

Superuser for Permissions to Run Client Proxies (-sysuser)

`-sysuser=`*`superuser_name/superuser_password`*

Use this 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 protocol. 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.

See "Additional PL/SQL Utility Scripts" on page A-21 for examples of scripts to grant and revoke permissions.

Option to Access SQLJ Functionality

This section documents the `-sqlj` option, which you can use to pass SQLJ options to the SQLJ translator (which JPublisher invokes) through the JPublisher command line.

Settings for the SQLJ Translator (-sqlj)

`-sqlj=`*`sqlj_options`*

In Oracle Database 10*g*, SQLJ translation is automatic by default when you run JPublisher, as discussed in "JPublisher Usage of the Oracle SQLJ Implementation" on page 1-4. Translation is transparent, with no visible `.sqlj` files resulting from JPublisher code generation.

For those familiar with SQLJ options and functionality, however, you can still specify SQLJ settings for the JPublisher invocation of the SQLJ translator. Use the JPublisher `-sqlj` option for this, as in the following example:

```
% jpub -user=scott/tiger -sqlj -optcols=true -optparams=true
       -optparamdefaults=datatype1(size1),datatype2(size)
```

Note the following:

- There is no "=" (equals 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 this 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 a setting of `-compatible=sqlj`, which skips the automatic SQLJ translation step and results in `.sqlj` output files from JPublisher. In this case, use no JPublisher options aside from `-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 line 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, which 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
- That 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

Also see "Backward Compatibility and Migration" on page 4-4.

Backward-Compatible Oracle Mapping for User-Defined Types (-compatible)

`-compatible={`**`oradata`**`|customdatum|both8i|8i|9i|sqlj}`

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 (classes representing SQL user-defined types).

or:

- Through a setting of `sqlj`, `8i`, `both8i`, or `9i`, you can specify a backward compatibility mode.

You can use one mode or the other, but not both.

Using -compatible to Specify an Interface If `-usertypes=oracle`, you have the option of setting `-compatible=customdatum` to implement the deprecated `CustomDatum` interface instead of the default `ORAData` interface in your generated classes for user-defined types. `CustomDatum` was replaced by `ORAData` in Oracle9*i*, but is still supported for backward compatibility.

The default setting is `oradata`, to use the `ORAData` interface. If `-usertypes=jdbc`, a `-compatible` setting of `customdatum` or `oradata` is ignored.

If you use JPublisher in a pre-Oracle9*i* environment, in which the `ORAData` interface is unsupported, 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 setting `sqlj`, `9i`, `8i`, or `both8i` to specify a backward compatibility mode.

The setting `-compatible=sqlj` instructs JPublisher to skip SQLJ translation and instead produce `.sqlj` files that you can work with directly. (In Oracle Database 10*g*, by default, SQLJ source files are automatically translated and deleted.) The `sqlj` setting has no effect on the generated code itself. To translate resulting `.sqlj` files, you can use the SQLJ translator directly (if available), or use the JPublisher `-sqlj` option. See "Option to Access SQLJ Functionality" on page 5-42.

The setting `-compatibility=9i` 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. See "Oracle9i Compatibility Mode" on page 4-7.

The setting `-compatible=8i` 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 a `-methods=always` setting, and generation of connection context declarations is equivalent to that for a `-context=generated` setting. See "Oracle8i Compatibility Mode" on page 4-7.

The option setting `-compatible=both8i` is for an alternative Oracle8*i* compatibility mode. With this setting, wrapper classes are generated to implement both the `ORAData` interface and the `CustomDatum` interface. Code is otherwise 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.

Note, however, that using `ORAData` requires an Oracle9*i* Release 1 (9.0.1) or higher 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 one (or more) of the types contains stored procedures, then by default JPublisher generates `.sqlj` files for all the SQL types, not just the types that have stored procedures. (But you have the option of explicitly suppressing the generation of SQLJ classes through the JPublisher `-methods=false` setting, which results in all non-SQLJ classes.)

Java Environment Options

This section discusses JPublisher options 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 code generated by JPublisher.
- The `-vm` option specifies the Java virtual machine (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—for call-outs and to map the `SYS.XMLType`—are unavailable.

Classpath for Translation and Compilation (-classpath)

`-classpath=`*`path1`*`:`*`path2`*`:...:`*`pathN`*

Use this option to specify the Java classpath for JPublisher to use in resolving Java source and classes during translation and compilation. The following (wraparound) command line 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 (-compiler-executable)

`-compiler-executable=`*`path_to_compiler_executable`*

Use this 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. See the next section, "Java Version (-vm)", for an example.

Java Version (-vm)

`-vm=`*`path_to_JVM_executable`*

Use this option if you want to use a Java virtual machine other than the JVM that JPublisher would use by default on your system. Specify the path to an alternative Java executable file.

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 (wraparound) command line 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
```

Code Generation for Wrapper Class and PL/SQL Wrapper Options

The following sections discuss code generation for the wrapper classes and PL/SQL wrappers produced in conjunction with the `-proxywsdl`, `-proxyclasses`, and `-proxyopts` options, usually for Web services call-outs.

See "Options to Facilitate Web Services Call-Outs" on page 5-34 for information about these options. See "Mechanisms Used in Exposing Java to PL/SQL" on page 5-36 for related concepts in how JPublisher produces wrapper classes, including the singleton mechanism and handle mechanism discussed shortly.

- Wrapper Class Generation without Handles
- Wrapper Class Generation with Handles
- Code Generation for Method Parameters
- Code Generation for Table Functions

Wrapper Class Generation without Handles

This section discusses how JPublisher generates wrapper classes for the use of instance methods in the singleton scenario, in which a single default class instance is used during execution and, therefore, the generated code does not use handles. Generation without handles is the default for the setting `-proxyopts=jaxrpc` or `-proxyopts=soap`, or it can be explicitly requested with the setting `-proxyopts=single`.

When JPublisher generates wrapper classes without handles, it takes the following actions:

- In each wrapper class (`BazJPub`, for example), JPublisher defines a default private static instance `DEFAULT_INSTANCE` of the class being wrapped (`foo.bar.Baz`, for example), initialized to `null`.
- For every method invocation of the wrapped class, JPublisher first checks whether the instance is `null`. If it is *not* `null`, then JPublisher uses the instance to invoke the method. If it *is* null, then JPublisher creates a new instance.
- JPublisher generates a `public static release`*`XXX`*`()` method (`releaseBaz()`, for example). Invoking this method sets the `DEFAULT_INSTANCE` field to `null`. Note that this behavior is different from that

of the `releaseAllXXX()` method (`releaseAllBaz()`, for example) that JPublisher provides when generating code with handles.

In addition, assume that the class being wrapped (`Baz`) contains the following instance methods, in which `X` is a user-defined type:

- `public void p(X)`
- `public X f()`

JPublisher generates methods in the wrapper class (`BazJPub`) as follows.

- JPublisher generates the following, if `X` is mappable to a SQL object:

  `public static void p(X arg)`

  Or it generates the following, if `X` is a bean:

  `public static void p(oracle.sql.STRUCT arg)`

  If `X` is neither a bean nor mappable to a SQL object, then neither method is generated.

  During program execution, the JPublisher runtime creates an instance of `foo.bar.Baz` into `DEFAULT_INSTANCE`, if this instance does not already exist, and calls the `p(arg)` method on `DEFAULT_INSTANCE`.

- And JPublisher generates the following, if `X` is mappable to a SQL object:

  `public static X f()`

  Or it generates the following, if `X` is a bean:

  `public static oracle.sql.STRUCT f()`

  If `X` is neither a bean nor mappable to a SQL object, then this wrapper method is not generated.

Wrapper Class Generation with Handles

This section discusses how JPublisher generates wrapper classes for the use of instance methods in the handle scenario, in which class instances are identified by handles (`long` values) during execution. In the generated wrapper classes, static methods are generated in place of instance methods, with the calling sequences modified to specify the appropriate handle. Generation with handles occurs with the setting `-proxyopts=multiple`. See "Java and PL/SQL Wrappers for Instance Methods Using the Handle Mechanism" on page A-24 for related code generation examples.

In the handle scenario, a generated class—`foo.bar.BazJPub`, wrapping the user class `Baz`, for example—always contains the following public static methods:

- `public static long newBaz()`

  Creates a new instance of `Baz` and returns a handle on this instance.

- `public static void releaseBaz(long handleBaz)`

  Releases the `Baz` instance associated with the specified handle.

- `public static void releaseAllBaz()`

  Releases all `Baz` instances that have been associated with any handle.

In addition, JPublisher generates the following public static methods to be used internally. These methods are *not* published to PL/SQL:

- `public static Baz _getBaz(long handleBaz)`

Retrieves the `Baz` instance associated with the specified handle.

- `public static long _setBaz(Baz baz)`

  Stores a `Baz` instance and returns a handle to it.

Additionally, assume that `Baz` contains the following instance methods, in which `X` is a user-defined Java type:

- `public void p(X)`
- `public X f()`

JPublisher generates the following Java methods in `BazJPub`:

- `public static void p(long handleBaz, X arg)`

  The implementation uses the handle to retrieve the `foo.bar.Baz` instance and then calls `p(arg)` on it. If the class `X` is not directly mappable to SQL but is a class to be wrapped, then JPublisher, instead, generates the following signature:

  `public static void p(long handleBaz, long handleX)`

  Otherwise, JPublisher cannot generate any wrapper method.

- `public static X f(long handleBaz)`

  The implementation uses the handle to retrieve the `foo.bar.Baz` instance and then calls `f()` on it. If the class `X` is not directly mappable to SQL but is a class to be wrapped, then JPublisher, instead, generates the following signature:

  `public static long f(long handleBaz)`

  This method returns the handle to the `X` instance.

Code Generation for Method Parameters

In generating Java wrapper classes, JPublisher can supply wrappers for methods with object or collection type parameters as well as providing wrappers for methods with primitive type parameters.

Assume a function such as the following:

```
X f(Y)
```

JPublisher uses the arguments directly if any of the following is true:

- The arguments are mappable to SQL.
- The arguments are being wrapped themselves.
- The arguments are array types.
- The arguments are JavaBeans.

Of particular interest for Web services are methods with argument types conforming to the JavaBeans specification. Inside Oracle Database, a JavaBeans type is mapped to the type `oracle.sql.STRUCT`. The PL/SQL wrapper generated on top of the wrapper class maps `oracle.sql.STRUCT` to a SQL object type generated automatically by JPublisher, with a set of attributes corresponding to the properties of the JavaBeans type.

Now the use of array and JavaBeans arguments is further discussed and demonstrated. Assume a function such as the following:

```
X f(Y[])
```

JPublisher will translate the method `f()` into three methods in the wrapper class:

```
X f(oracle.sql.ARRAY a);
X f_o(Y[] a);
X f_io(Y[] a);
```

The first method treats the array argument as an input. The second treats the array as a holder for an output value. The third treats the array as a holder for both input and output values. These methods are reflected in the following PL/SQL stored procedures that wrap the generated Java wrapper:

```
FUNCTION F (a Y_ARRAY) RETURN XSQL;
FUNCTION F (a OUT YSQL) RETURN XSQL;
FUNCTION F (a IN OUT YSQL) RETURN XSQL;
```

`XSQL`, `YSQL` and `Y_ARRAY` are the SQL types that map to `X`, `Y`, and `Y[]` (array of `Y`).

For JAX-RPC Web services, the client proxy class uses holders to represent `OUT` or `IN OUT` arguments. (See "Passing Output Parameters in JAX-RPC Holders" on page 3-3.) In this case, JPublisher recognizes arguments that are holders, such as in the following:

```
void p(A,B,CHolder,DHolder)
```

Whenever holder classes are used, JPublisher produces Java wrapper code that maps from `X` to `XHolder` and back. JPublisher assumes that all holders are `IN OUT`. The corresponding Java and PL/SQL wrappers will be the following, respectively:

```
void p(A x, B y, C[] z, D[] w)
```

and:

```
PROCEDURE p(x ASQL, y BSQL, x IN OUT CSQL, w IN OUT DSQL);
```

`ASQL`, `BSQL`, `CSQL`, and `DSQL` are the SQL types that map to `A`, `B`, `C`, and `D`.

Note: Only JAX-RPC holders are currently supported.

Now consider the following two classes. `CalloutX` references `CalloutY`, a JavaBean, as a parameter type. Additionally, methods in `CalloutX` use array arguments.

```
public class CalloutX
{
   public void add(CalloutY arg)   { tot = tot + arg.getTotal(); }
   public static int[] add(int[] i,  int[] j)
   {
     int[] r = new int[i.length];
     for (int k=0; k<i.length; k++) r[k] = i[k] + j[k];
     return r;
   }
}
public class CalloutY
{
   public void setTotal(int total)    { this.total = total; }
   public int getTotal() { return total; }
   private int total;
}
```

Compile the two classes and make sure they are both in your classpath. Then run JPublisher as follows.

```
% jpub -proxyclasses=CalloutX -proxyopts=single
```

The generated wrapper class, `CalloutXJPub`, will have the following methods:

```
public static void add(oracle.sql.STRUCT __jPt_0);
public static oracle.sql.ARRAY add
                             (oracle.sql.ARRAY __jPt_0,oracle.sql.ARRAY __jPt_1);
public static oracle.sql.ARRAY add_o(int[] p0,int[] p1);
public static oracle.sql.ARRAY add_io(int[] p0,int[] p1);
```

The first `add()` method in `CalloutXJPub` corresponds to the `add(CalloutY)` method in `CalloutX`. Because `CalloutY` is a JavaBean, JPublisher maps it to `oracle.sql.STRUCT`, as discussed earlier in this section.

The second `add()` method, the `add_o()` method, and the `add_io()` method in `CalloutXJPub` correspond to the `add(int[], int[])` method in `CalloutX`. This demonstrates what was discussed earlier in this section, regarding the three wrapper methods that JPublisher generates for a method with an array parameter. The type `oracle.sql.ARRAY` is used for input, in the second `add()` method. The original array type, here `int[]`, is used for output or for input/output—in the `add_o()` method and the `add_io()` method, respectively.

The PL/SQL wrapper script generated by JPublisher specifies creation of SQL types and PL/SQL call specs, to map to the `CalloutXJPub` class, as shown immediately below. The first `CREATE` statement is for an object type to correspond to the Java `STRUCT` type. The second `CREATE` statement is for a SQL TABLE type (collection type) to correspond to the Java `ARRAY` type. The third `CREATE` statement is for the PL/SQL call specs for the two `add()` methods, the `add_o()` method, and the `add_io()` method.

```
CREATE OR REPLACE TYPE SQLCalloutY AS OBJECT (total NUMBER);
/
CREATE OR REPLACE TYPE S_NUMBER AS TABLE OF NUMBER;
/
CREATE OR REPLACE PACKAGE JPUB_PLSQL_WRAPPER AS
   PROCEDURE add(p0 SQLCalloutY);
   FUNCTION add(p0 S_NUMBER,p1 S_NUMBER) RETURN S_NUMBER;
   FUNCTION add_o(p0 OUT NUMBER,p1 OUT NUMBER) RETURN S_NUMBER;
   FUNCTION add_io(p0 IN OUT NUMBER,p1 IN OUT NUMBER) RETURN S_NUMBER;
END JPUB_PLSQL_WRAPPER;
/
```

See "Java and PL/SQL Wrappers for Static Methods" on page A-23 for a related code generation example.

Code Generation for Table Functions

This section discusses the use of PL/SQL table functions in code generated by JPublisher, to return data for Web services operations. There is also a link to an example in the appendix for further details. 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.

Note: See the *PL/SQL User's Guide and Reference* for general information about table functions.

Use the JPublisher setting `-proxyopts=jaxrpc,tabfun` or `-proxyopts=soap,tabfun` to request this behavior. See "Settings for Java and

PL/SQL Wrapper Generation (-proxyopts)" on page 5-38 for information about that option.

For a table function to be generated for a given Web Service operation, the following must be true:

- For wrapping instance methods, the singleton mechanism must be enabled. This is the case whenever `jaxrpc` (the default) or `soap` is included in the `-proxyopts` setting. Both of these imply `single` mode.
- The wrapped Web service method must correspond to a stored procedure with `OUT` arguments or to a stored function.

The term *graph* is used with table functions. In this usage, a graph is a SQL object that essentially 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 Web service and the output from that Web service.

As an example, consider the following, which defines the graph object `GRAPH_getProperty`, the table `GRAPH_TAB_getProperty` of graph objects, and the table function `TO_TABLE_getProperty` that returns the table of graph objects:

```
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;
/
FUNCTION TO_TABLE_getProperty(cur SYS_REFCURSOR) RETURN GRAPH_TAB_getProperty
PIPELINED;
```

Also note that a table function always takes a REF CURSOR as input. In this example, demonstrated fully in "Generated Code: Java and PL/SQL Wrappers for Web Services" on page A-10, the `getProperty` operation of the Web service takes a system property name as input and returns the value of that property. For example, if the Web service runs on a Sun Microsystems Solaris system, the `geProperty` operation with input "`os.name`" returns "`SunOS`". The REF CURSOR has one column, consisting of system property names. The graph defines two columns of character strings—one for property names and one for corresponding property values. The table of graph objects combines the system property names from the REF CURSOR, which are input to `getProperty`, and the corresponding property values, which are output by `getProperty`. The table function returns this table, with a property name and the corresponding value in each row.

JPublisher Input Files

These 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. (And `-props` is the only option that you cannot specify in a properties file.)

In a properties file, enter one (and only one) option with its associated value on each line. Enter each option setting with the following prefix (including the period), case-sensitive:

```
jpub.
```

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 "`--`" (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.

Additionally, you can use line continuation to spread a JPublisher option over several lines in the properties file. A line to be continued must have "`\`" (backslash character) as the last character, immediately after the text of the line. Any leading space or "`--`" 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
```

Because of 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 line (a single wraparound line):

```
% jpub -user=scott/tiger -sql=employee -mapping=oracle -case=lower -package=corp
       -dir=demo
```

Now consider the following:

```
% jpub -props=my_properties
```

This command line is equivalent to the first example if you assume `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 (including the period) at the beginning of each option name. If you enter anything other than white space or "`--`" 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, as for "`Jpub.notreally=a jpub option`".

"JPublisher Options" on page 5-1 describes all the JPublisher options.

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, `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`***`)`**, a comma-delimited list of names, enclosed in parentheses, to be published. 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.

Important: If a user-defined type was defined in a case-sensitive way (in quotes) in SQL, then you must specify the name in quotes. For example:

```
SQL "CaseSenstiveType" AS CaseSensitiveType
```

Alternatively, you can also specify a non-case-sensitive schema name:

```
SQL SCOTT."CaseSensitiveType" AS CaseSensitiveType
```

You can also specify a case-sensitive schema name:

```
SQL "Scott"."CaseSensitiveType" AS CaseSensitiveType
```

The `AS` clauses, described shortly, are optional.

Avoid situations where a dot (".") is part of the schema name or 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, referred to as the base class, and the `AS` clause specifies the name of a Java class, referred to as the user subclass, that extends the generated base class. 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, not to the base class. If you later run the same JPublisher command to republish the SQL type, the generated class is overwritten but the user subclass is not. (Also see "Extending JPublisher-Generated Classes" on page 3-14.)

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 override the `-case` option. For more information on how to name packages, see "Package Naming Rules in the INPUT File" on page 5-55.

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 (stored procedure) 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. Its case 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 (modified according to the setting of the `-case` option, if applicable). 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, non-qualified class name in the `INPUT` file, then the fully qualified class name includes the package name from the `-package` option, if applicable. 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.

For more examples of how the package name is determined, see "Name for Generated Java Package (-package)" on page 5-29.

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; otherwise, the `INPUT` file setting takes precedence.

Translating Additional Types It may be necessary for JPublisher to translate additional types 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. Recall the example in "Sample JPublisher Translation" on page 1-27. Assume that the object type definition for `EMPLOYEE` had included an attribute called `ADDRESS`, and `ADDRESS` was 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 1-27. Consider the following command line (a single wraparound line):

```
% 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 upper case 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 upper case, 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 silently overwrites the first. For example, if the `INPUT` file contains:

```
type PERSON1 as Person
type PERSON2 as Person
```

JPublisher creates the file `Person.java` for `PERSON1` and then overwrites it for type `PERSON2`.

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, 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 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 do not fit conveniently into the corresponding sections earlier in the manual:

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 3-16. 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.

To review, 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`, 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 the following:

- 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 style file `webservices10.properties`, for use in Web services. (See "Support for XMLTYPE" on page 2-12.)
- The holder class for `Source` data, `javax.xml.rpc.holders.SourceHolder`, is used for the output `Source` argument. (See "Passing Output Parameters in JAX-RPC Holders" on page 3-3.)
- The style file specifies the generated code naming pattern "`%2Base:%2User#%2`" relating to the JPublisher command "`-s foo_pack:FooPack`", which results in

generation of interface code in `FooPack.java`, base class code in `FooPackBase.java`, and user subclass code in `FooPackUser.java`.

The wrapper method `foo()` in the user subclass `FooPackUser` uses the following type transformation functionality and a call to the corresponding `_foo()` method of the generated base class, which is where the 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

This is code 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 etc. is 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

This is code 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.

Comments indicate corresponding SQLJ code, which JPublisher translates automatically during 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 etc. is 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

This is code for the user subclass that JPublisher generates in `FooPackUser.java`. 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.

This class extends the class `FooPackBase` and implements the interface `FooPack`.

```
import java.sql.SQLException;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;
// Ensure that the java.io.* package etc. is 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, relating to the discussion in "Declaration of SQL Statements to Translate (-sqlstatement)" on page 5-12.

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 the following 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.)

Here is the translated SQLJ code that JPublisher would produce:

```
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: Java and PL/SQL Wrappers for Web Services

This section contains code examples for JAX-RPC client proxies and the associated Java and PL/SQL wrappers, generated according to the WSDL document indicated by the `-proxywsdl` setting. The code in this example is for use with an EJB Web service, and also uses a table function. The following are included:

- WSDL document

  JPublisher accesses the WSDL document, the content of which determines the JAX-RPC client proxy classes to generate.

- JAX-RPC client proxy stub class, an associated factory class, and the interfaces they implement

  These classes and interfaces are generated by the Oracle Database Web services assembler tool, which is invoked by JPublisher. Resulting proxy classes and interfaces are loaded into the database. The client proxy stub class is used for invoking the Web service.

- Java wrapper class and PL/SQL wrapper

  These wrappers are generated by JPublisher for use in the database. Use the PL/SQL wrapper to access the client proxy class from PL/SQL. The client proxy class, in turn, invokes the Web service. The wrapper class is a required intermediate layer to publish instance methods of the client proxy class as static methods, because PL/SQL supports only static methods.

- Associated PL/SQL utility scripts

  These scripts are generated by JPublisher. The scripts are to create the PL/SQL wrapper in the database schema, grant permission to execute it, revoke that permission, and drop the PL/SQL wrapper from the database schema.

For this 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. Here is the 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
       -endpoint=http://localhost:8888/javacallout/javacallout
       -proxyopts=jaxrpc,tabfun -package=javacallout -dir=genproxy
```

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 execute the wrapper script.

See "WSDL Document for Java and PL/SQL Wrapper Generation (-proxywsdl)" on page 5-40 and "Web Services Endpoint (-endpoint)" on page 5-41 for information about those options. The `-endpoint` setting is used in the Java wrapper class that JPublisher generates, shown in "Java Wrapper Class and PL/SQL Wrapper" on page A-18. See "Code Generation for Table Functions" on page 5-50 for general information about table functions.

For more Java and PL/SQL wrapper examples, also see "Generated Code: Java and PL/SQL Wrappers for General Use" on page A-22.

WSDL Document

This section contains the WSDL document for the Web service. `HelloServiceInf` in the `<message>` element is the name of the service bean and determines the name of the interface that is generated and that is 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.

```
<?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>
```

JAX-RPC Client Proxy Classes and Interfaces

This section shows the JAX-RPC client proxy stub class, the interface that it implements, a factory class to produce instances of the stub class, and the interface that the factory class implements. These classes and interfaces are all generated by the Oracle Database Web services assembler tool, which is invoked by JPublisher. The stub class is for invoking the Web service, through Java and PL/SQL wrappers that JPublisher produces.

Interface for JAX-RPC Client Proxy Stub Class

Here is code for the interface, `HelloServiceInf`, implemented by the JAX-RPC client proxy stub class, `HelloServiceInf_Stub`, which is shown immediately following.

```
// !DO NOT EDIT THIS FILE!
// This source file is generated by Oracle tools
// Contents may be subject to change
// For reporting problems, use the following
// Version = [failed to localize] wscompile.version(Oracle WebServices, 10i, J1)

package javacallout;

public interface HelloServiceInf extends java.rmi.Remote {
    public java.lang.String getProperty(java.lang.String string_1)
                                    throws java.rmi.RemoteException;
}
```

JAX-RPC Client Proxy Stub Class

Here is code for the JAX-RPC client proxy stub class, `HelloServiceInf_Stub`. It implements the interface `HelloServiceInf`, shown immediately preceding.

```
// !DO NOT EDIT THIS FILE!
// This source file is generated by Oracle tools
// Contents may be subject to change
// For reporting problems, use the following
// Version = [failed to localize] wscompile.version(Oracle WebServices, 10i, J1)

package javacallout;

import oracle.j2ee.ws.server.MessageContextProperties;
import oracle.j2ee.ws.common.streaming.*;
import oracle.j2ee.ws.common.encoding.*;
import oracle.j2ee.ws.common.encoding.soap.SOAPConstants;
import oracle.j2ee.ws.common.encoding.soap.SOAP12Constants;
import oracle.j2ee.ws.common.encoding.literal.*;
import oracle.j2ee.ws.common.soap.streaming.*;
import oracle.j2ee.ws.common.soap.message.*;
import oracle.j2ee.ws.common.soap.SOAPVersion;
import oracle.j2ee.ws.common.soap.SOAPEncodingConstants;
import oracle.j2ee.ws.common.wsdl.document.schema.SchemaConstants;
import javax.xml.namespace.QName;
import java.rmi.RemoteException;
import java.util.Iterator;
import java.lang.reflect.*;
import oracle.j2ee.ws.client.SenderException;
import oracle.j2ee.ws.client.*;
import oracle.j2ee.ws.client.http.*;
import javax.xml.rpc.handler.*;
import javax.xml.rpc.JAXRPCException;
import javax.xml.rpc.soap.SOAPFaultException;

public class HelloServiceInf_Stub
    extends oracle.j2ee.ws.client.StubBase
    implements javacallout.HelloServiceInf {

    /*
     *  public constructor
     */
    public HelloServiceInf_Stub(HandlerChain handlerChain) {
        super(handlerChain);
        _setProperty(ENDPOINT_ADDRESS_PROPERTY,
                     "http://localhost:8888/javacallout/javacallout");
    }

    /*
     *  implementation of getProperty
     */
    public java.lang.String getProperty(java.lang.String string_1)
        throws java.rmi.RemoteException {

        try {

            StreamingSenderState _state = _start(_handlerChain);

            InternalSOAPMessage _request = _state.getRequest();
            _request.setOperationCode(getProperty_OPCODE);
            javacallout.HelloServiceInf_getProperty_RequestStruct
                 _myHelloServiceInf_getProperty_RequestStruct =
                 new javacallout.HelloServiceInf_getProperty_RequestStruct();

            _myHelloServiceInf_getProperty_RequestStruct.setString_1(string_1);

            SOAPBlockInfo _bodyBlock =
                 new SOAPBlockInfo(ns1_getProperty_getProperty_QNAME);
            _bodyBlock.setValue(_myHelloServiceInf_getProperty_RequestStruct);
            _bodyBlock.setSerializer
                 (myHelloServiceInf_getProperty_RequestStruct_SOAPSerializer);
            _request.setBody(_bodyBlock);

            _state.getMessageContext().setProperty("http.soap.action", "");

            _send((String) _getProperty(ENDPOINT_ADDRESS_PROPERTY), _state);

            javacallout.HelloServiceInf_getProperty_ResponseStruct
                 _myHelloServiceInf_getProperty_ResponseStruct = null;
            Object _responseObj = _state.getResponse().getBody().getValue();
            if (_responseObj instanceof SOAPDeserializationState) {
                _myHelloServiceInf_getProperty_ResponseStruct =
                    (javacallout.HelloServiceInf_getProperty_ResponseStruct)
                        ((SOAPDeserializationState)_responseObj).getInstance();
            } else {
                _myHelloServiceInf_getProperty_ResponseStruct =
            (javacallout.HelloServiceInf_getProperty_ResponseStruct)_responseObj;
            }

            return _myHelloServiceInf_getProperty_ResponseStruct.getResult();
        } catch (RemoteException e) {
            // let this one through unchanged
            throw e;
        } catch (JAXRPCException e) {
            throw new RemoteException(e.getMessage(), e);
        } catch (Exception e) {
            if (e instanceof RuntimeException) {
                throw (RuntimeException)e;
            } else {
                throw new RemoteException(e.getMessage(), e);
            }
        }
    }

    /*
     *  this method deserializes the request/response structure in the body
     */
    protected void _readFirstBodyElement(XMLReader bodyReader,
              SOAPDeserializationContext deserializationContext,
              StreamingSenderState  state) throws Exception {
        int opcode = state.getRequest().getOperationCode();
        switch (opcode) {
            case getProperty_OPCODE:
                _deserialize_getProperty(bodyReader, deserializationContext,
                                         state);
                break;
            default:
                throw new SenderException("sender.response.unrecognizedOperation",
                                           Integer.toString(opcode));
        }
    }

    /*
     * This method deserializes the body of the getProperty operation.
     */
    private void _deserialize_getProperty(XMLReader bodyReader,
            SOAPDeserializationContext deserializationContext,
            StreamingSenderState state) throws Exception {
        Object myHelloServiceInf_getProperty_ResponseStructObj =
          myHelloServiceInf_getProperty_ResponseStruct_SOAPSerializer.deserialize
               (ns1_getProperty_getPropertyResponse_QNAME,
                bodyReader, deserializationContext);

        SOAPBlockInfo bodyBlock =
             new SOAPBlockInfo(ns1_getProperty_getPropertyResponse_QNAME);
        bodyBlock.setValue(myHelloServiceInf_getProperty_ResponseStructObj);
        state.getResponse().setBody(bodyBlock);
    }

    public String _getEncodingStyle() {
        return SOAPNamespaceConstants.ENCODING;
    }

    public void _setEncodingStyle(String encodingStyle) {
        throw new UnsupportedOperationException("cannot set encoding style");
    }
    /*
     * This method returns an array containing (prefix, nsURI) pairs.
     */
    protected String[] _getNamespaceDeclarations() {
        return myNamespace_declarations;
    }

    /*
     * This method returns an array containing the names of the headers
     * we understand.
     */
    public QName[] _getUnderstoodHeaders() {
        return understoodHeaderNames;
    }

    public void _initialize(InternalTypeMappingRegistry registry)
        throws Exception {
        super._initialize(registry);
        myHelloServiceInf_getProperty_RequestStruct_SOAPSerializer =
            (CombinedSerializer)registry.getSerializer
            (SOAPConstants.NS_SOAP_ENCODING,
            javacallout.HelloServiceInf_getProperty_RequestStruct.class,
            ns1_getProperty_TYPE_QNAME);
        myHelloServiceInf_getProperty_ResponseStruct_SOAPSerializer =
            (CombinedSerializer)registry.getSerializer
            (SOAPConstants.NS_SOAP_ENCODING,
            javacallout.HelloServiceInf_getProperty_ResponseStruct.class,
            ns1_getPropertyResponse_TYPE_QNAME);
    }

    private static final QName _portName =
      new QName("http://oracle.j2ee.ws/javacallout/Hello", "HelloServiceInfPort");
    private static final int getProperty_OPCODE = 0;
    private static final QName ns1_getProperty_getProperty_QNAME =
      new QName("http://oracle.j2ee.ws/javacallout/Hello", "getProperty");
    private static final QName ns1_getProperty_TYPE_QNAME =
      new QName("http://oracle.j2ee.ws/javacallout/Hello", "getProperty");
    private CombinedSerializer
            myHelloServiceInf_getProperty_RequestStruct_SOAPSerializer;
    private static final QName ns1_getProperty_getPropertyResponse_QNAME =
      new QName("http://oracle.j2ee.ws/javacallout/Hello", "getPropertyResponse");
    private static final QName ns1_getPropertyResponse_TYPE_QNAME =
      new QName("http://oracle.j2ee.ws/javacallout/Hello", "getPropertyResponse");
    private CombinedSerializer
            myHelloServiceInf_getProperty_ResponseStruct_SOAPSerializer;
    private static final String[] myNamespace_declarations =
                                    new String[] {
                                    "ns0",
                                    "http://oracle.j2ee.ws/javacallout/Hello"
                                    };

    private static final QName[] understoodHeaderNames = new QName[] {  };
}
```

Interface for JAX-RPC Client Proxy Factory Class

Here is the code for the interface, `HelloServiceEJB`, implemented by the client proxy factory class, `HelloServiceEJB_Impl`, which is shown immediately following.

```
// !DO NOT EDIT THIS FILE!
// This source file is generated by Oracle tools
// Contents may be subject to change
// For reporting problems, use the following
// Version = [failed to localize] wscompile.version(Oracle WebServices, 10i, J1)

package javacallout;

import javax.xml.rpc.*;

public interface HelloServiceEJB extends javax.xml.rpc.Service {
    public javacallout.HelloServiceInf getHelloServiceInfPort();
}
```

JAX-RPC Client Proxy Factory Class

Here is code for the client proxy factory class, `HelloServiceEJB_Impl`, used to create instances of the client proxy stub class, `HelloServiceInf_Stub`, which is shown earlier in this section. The `HelloServiceEJB_Impl` class implements the `HelloServiceEJB` interface, shown immediately preceding.

```
// !DO NOT EDIT THIS FILE!
// This source file is generated by Oracle tools
// Contents may be subject to change
// For reporting problems, use the following
// Version = null

package javacallout;

import oracle.j2ee.ws.common.encoding.*;
import oracle.j2ee.ws.client.ServiceExceptionImpl;
import oracle.j2ee.ws.common.util.exception.*;
import oracle.j2ee.ws.common.soap.SOAPVersion;
import oracle.j2ee.ws.client.HandlerChainImpl;
import javax.xml.rpc.*;
import javax.xml.rpc.encoding.*;
import javax.xml.rpc.handler.HandlerChain;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.namespace.QName;

public class HelloServiceEJB_Impl extends oracle.j2ee.ws.client.BasicService
implements HelloServiceEJB {
    private static final QName serviceName =
      new QName("http://oracle.j2ee.ws/javacallout/Hello", "HelloServiceEJB");
    private static final QName ns1_HelloServiceInfPort_QNAME =
      new QName("http://oracle.j2ee.ws/javacallout/Hello", "HelloServiceInfPort");
    private static final Class helloServiceInf_PortClass =
                              javacallout.HelloServiceInf.class;

    public HelloServiceEJB_Impl() {
        super(serviceName, new QName[] {
                        ns1_HelloServiceInfPort_QNAME
                    },
            new javacallout.HelloServiceEJB_SerializerRegistry().getRegistry());

    }

    public java.rmi.Remote getPort(QName portName, Class serviceDefInterface)
                                 throws javax.xml.rpc.ServiceException {
        try {
            if (portName.equals(ns1_HelloServiceInfPort_QNAME) &&
                serviceDefInterface.equals(helloServiceInf_PortClass)) {
                return getHelloServiceInfPort();
            }
        } catch (Exception e) {
            throw new ServiceExceptionImpl(new LocalizableExceptionAdapter(e));
        }
        return super.getPort(portName, serviceDefInterface);
    }

    public java.rmi.Remote getPort(Class serviceDefInterface)
                               throws javax.xml.rpc.ServiceException {
        try {
            if (serviceDefInterface.equals(helloServiceInf_PortClass)) {
                return getHelloServiceInfPort();
            }
        } catch (Exception e) {
            throw new ServiceExceptionImpl(new LocalizableExceptionAdapter(e));
        }
        return super.getPort(serviceDefInterface);
    }

    public javacallout.HelloServiceInf getHelloServiceInfPort() {
        String[] roles = new String[] {};
        HandlerChainImpl handlerChain =
           new HandlerChainImpl
           (getHandlerRegistry().getHandlerChain(ns1_HelloServiceInfPort_QNAME));
        handlerChain.setRoles(roles);
        javacallout.HelloServiceInf_Stub stub =
             new javacallout.HelloServiceInf_Stub(handlerChain);
        try {
            stub._initialize(super.internalTypeRegistry);
        } catch (JAXRPCException e) {
            throw e;
        } catch (Exception e) {
            throw new JAXRPCException(e.getMessage(), e);
        }
        return stub;
    }
}
```

Java Wrapper Class and PL/SQL Wrapper

This section shows code for the Java wrapper class and PL/SQL wrapper that JPublisher generates. These are used to call the proxy client stub class from PL/SQL.

Wrapper Class

Here is the class, `HelloServiceEJBJPub`, that JPublisher generates as a Java wrapper for the proxy client stub class. The Java wrapper is required to publish instance methods of the stub class as static methods, to allow accessibility from PL/SQL. The Web service endpoint in the code, "`http://localhost:8888/javacallout/javacallout`", is according to the `-endpoint` setting in the JPublisher command line.

The corresponding PL/SQL wrapper is shown immediately following.

```
package javacallout;
import javax.xml.rpc.Stub;

public class HelloServiceEJBJPub
{
   public static void release()
   {
      m_service = null;
      m_port0 = null;
   }

   private static void init()
   {
      m_service = new javacallout.HelloServiceEJB_Impl();
      m_port0 = (javacallout.HelloServiceInf)(m_service.getHelloServiceInfPort());
      ((Stub)m_port0)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
                     "http://localhost:8888/javacallout/javacallout");
   }

   private static javacallout.HelloServiceEJB_Impl m_service;
   private static javacallout.HelloServiceInf m_port0 = null;

   public static java.lang.String getProperty(java.lang.String p0)
   throws java.rmi.RemoteException
   {
     try
     {
     if (m_port0==null) init();
      java.lang.String o;
      o = m_port0.getProperty(p0);
      return o;
     }
     catch (Exception e)
     {
         throw new java.rmi.RemoteException(e.getMessage());
     }
   }
}
```

PL/SQL Wrapper

Here is the PL/SQL wrapper script, `plsql_wrapper.sql`, that creates the PL/SQL wrapper package, `JPUB_PLSQL_WRAPPER`. This package is created for invoking the Web service from PL/SQL. Note that it includes the definition of a table function—data from the Web service is returned through a database table, rather than through a normal function return.

An intermediate Java wrapper class, shown immediately preceding, is required to publish instance methods of the JAX-RPC client proxy stub class as static methods, which is required for accessibility from PL/SQL.

```
-- SQL types assisting the procedures that invoke web services
CREATE OR REPLACE TYPE GRAPH_getProperty AS OBJECT(p0 VARCHAR2(32767),
                                                   res VARCHAR2(32767));
/

SHOW ERRORS
CREATE OR REPLACE TYPE GRAPH_TAB_getProperty AS TABLE OF GRAPH_getProperty;
/

SHOW ERRORS

-- 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;
/

SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY JPUB_PLSQL_WRAPPER IS
   FUNCTION getProperty(p0 VARCHAR2) RETURN VARCHAR2
   as language java
   name 'javacallout.HelloServiceEJBJPub.getProperty(java.lang.String)
         return java.lang.String';
FUNCTION TO_TABLE_getProperty(cur SYS_REFCURSOR) RETURN GRAPH_TAB_getProperty
  PIPELINED IS
  p0 VARCHAR2(32767);
  res VARCHAR2(32767);
BEGIN
  LOOP
    FETCH cur INTO p0;
    EXIT WHEN cur%NOTFOUND;
    res:=getProperty(p0);
    PIPE ROW(GRAPH_getProperty(p0,res));
  END LOOP;
  CLOSE cur;
  RETURN;
END TO_TABLE_getProperty;

END JPUB_PLSQL_WRAPPER;
/
SHOW ERRORS

EXIT;
```

Because a `-user` setting is 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 ("`SunOS`"):

```
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`"). Then `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. Following is a sample usage of `TO_TABLE_getProperty`.

```
SQL> -- Test Table Function
SQL> create table props (name varchar2(50));
Table created.

SQL> insert into props values('os.version');
1 row created.

SQL> insert into props values('java.version');
1 row created.
SQL> insert into props values('file.separator');
1 row created.

SQL> insert into props values('file.encoding.pkg');
1 row created.

SQL> insert into props values('java.vm.info');
1 row created.

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. For example, you see that the operating system is Sun Microsystems Solaris 5.8.

See "Code Generation for Table Functions" on page 5-50 for general information about table functions.

Additional PL/SQL Utility Scripts

This section illustrates PL/SQL utility scripts that JPublisher generates, in addition to the wrapper script, plsql_wrapper.sql, already shown. There are scripts to grant permission for execution of the PL/SQL wrapper, revoke permission, and drop the PL/SQL package from the database schema.

See "Superuser for Permissions to Run Client Proxies (-sysuser)" on page 5-42 for information about the option to specify a superuser name and password for running these scripts.

Script to Grant Permission for PL/SQL Wrapper

Here is the script to grant permission to execute JPUB_PLSQL_WRAPPER.

```
-- Run this script as SYSDBA to grant permissions for SCOTT to run the PL/SQL
-- wrapper procedures.
BEGIN
dbms_java.grant_permission('SCOTT', 'SYS:java.lang.RuntimePermission',
                           'accessClassInPackage.sun.util.calendar', '' );
dbms_java.grant_permission('SCOTT', 'SYS:java.lang.RuntimePermission',
                           'getClassLoader', '' );
dbms_java.grant_permission('SCOTT', 'SYS:java.net.SocketPermission', '*',
                           'connect,resolve' );
END;
```

```
/

EXIT
```

Script to Revoke Permission for PL/SQL Wrapper

Here is the script to revoke permission to execute `JPUB_PLSQL_WRAPPER`.

```
-- Run this script as SYSDBA to revoke the permissions granted to SCOTT.
BEGIN
dbms_java.revoke_permission('SCOTT', 'SYS:java.lang.RuntimePermission',
                            'accessClassInPackage.sun.util.calendar', '' );
dbms_java.revoke_permission('SCOTT', 'SYS:java.lang.RuntimePermission',
                            'getClassLoader', '' );
dbms_java.revoke_permission('SCOTT', 'SYS:java.net.SocketPermission', '*',
                            'connect,resolve' );
END;
/

EXIT
```

Script to Drop the PL/SQL Package

Here is the script to drop `JPUB_PLSQL_WRAPPER` from the database schema:

```
-- Drop the PL/SQL procedures that invoke Web services.

DROP PACKAGE JPUB_PLSQL_WRAPPER;
EXIT
```

Generated Code: Java and PL/SQL Wrappers for General Use

This section has code examples for Java and PL/SQL wrappers generated for server-side Java classes, according to `-proxyclasses` and `-proxyopts` settings. See "Options to Facilitate Web Services Call-Outs" on page 5-34 for information about these and related options.

Assume that the Java and PL/SQL wrappers in these examples are for general use, not necessarily associated with Web services.

The following sections show the classes to be wrapped, the wrapper classes and PL/SQL wrappers produced for static methods, and the wrapper classes and PL/SQL wrappers produced for instance methods in the handle scenario.

- Classes to Be Wrapped
- Java and PL/SQL Wrappers for Static Methods
- Java and PL/SQL Wrappers for Instance Methods Using the Handle Mechanism

Also see "Generated Code: Java and PL/SQL Wrappers for Web Services" on page A-10.

Classes to Be Wrapped

This section shows classes for which Java and PL/SQL wrappers, shown in subsequent sections, will be generated. Assume that the following classes, named `X` and `Y`, are installed in the server:

```
public class X
 {
   public void add(int arg) { tot = tot + arg; }
   public void add(Y arg)   { tot = tot + arg.getTotal(); }
```

```
  public int getTot() { return tot; }
  public Y cloneTot()
  { Y y=new Y(); y.setTotal(getTot()); return y; }

  public static int add(int i, int j) { return i+j; }
  public static Y add(Y i, Y j)
  { Y y = new Y();
    y.setTotal(i.getTotal()+j.getTotal());
    return y;
  }
  private int tot;
}


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

Java and PL/SQL Wrappers for Static Methods

Assume that class X from the preceding section is processed by JPublisher with the following settings:

```
% jpub -proxyclasses=X -proxyopts=static
```

With these settings, JPublisher generates Java (as necessary) and PL/SQL wrappers for static methods only. This always includes a PL/SQL wrapper, and in this example also includes a wrapper class, XJPub, for the class X. A wrapper class is necessary, even when there are no instance methods to wrap, because class X uses the Java type Y for an argument in a method calling sequence. A wrapper class is required whenever types other than Java primitive types are used in method calls. In this case, the wrapper class has a wrapper method for the following:

```
add(Y arg) {...}
```

Note that because Y is a JavaBean type, it is mapped to the weakly typed oracle.sql.STRUCT class. See "Code Generation for Method Parameters" on page 5-48.

Wrapper Class

Here is the wrapper class:

```
import java.util.Hashtable;
import java.sql.StructDescriptor;
import java.sql.Connection;
import java.sql.DriverManager;
import oracle.sql.STRUCT;
import java.math.BigDecimal;

public class XJPub
{
  public static STRUCT add( STRUCT arg0, STRUCT arg1)
  throws java.sql.SQLException
  {
     Y _p1 = new Y();
     BigDecimal _p0 = (BigDecimal) arg0.getAttributes()[0];
     _p1.setTotal(_p0.intValue());
```

```
         Y _p3 = new Y();
         BigDecimal _p4 = (BigDecimal) arg2.getAttributes()[0];
         _p3.setTotal(_p4.intValue());
         Y _p5 = X.add(_p0, _p4);

         Connection conn = _getConnection();
         StructDescriptor structdesc =
                               StructDescriptor.createDescriptor("YSQL", conn);
         STRUCT _r = new STRUCT(structdesc, conn, new Object[]{_p5.getTotal()});
         return _r;
      }
      Connection _getConnection() throws java.sql.SQLException
      {
          return DriverManager.getConnection("jdbc:default:connection:");
      }
   }
```

PL/SQL Wrapper

Following is the PL/SQL wrapper. The type `YSQL` is created to enable the PL/SQL code to access `Y` instances.

```
CREATE OR REPLACE TYPE YSQL AS OBJECT(total number);
 /
 CREATE OR REPLACE PACKAGE JPUB_PLSQL_WRAPPER AS
   FUNCTION add(arg0 NUMBER, arg1 NUMBER) RETURN NUMBER;
   FUNCTION add(arg0 Y, arg1 Y) RETURN Y;
 END JPUB_PLSQL_WRAPPER;
 /
 CREATE OR REPLACE PACKAGE BODY JPUB_PLSQL_WRAPPER AS
   FUNCTION add(arg0 NUMBER, arg1 NUMBER) RETURN NUMBER AS LANGUAGE JAVA
   NAME 'X.add(int,int) return int';
   FUNCTION add(arg0 YSQL, arg1 YSQL) RETURN NUMBER AS LANGUAGE JAVA
   NAME 'XJPub.add(oracle.sql.STRUCT,oracle.sql.STRUCT) return oracle.sql.STRUCT';
 END JPUB_PLSQL_WRAPPER;
 /
```

Java and PL/SQL Wrappers for Instance Methods Using the Handle Mechanism

Assume that classes `X` and `Y` from "Classes to Be Wrapped" on page A-22 are processed by JPublisher with the following settings:

```
% jpub -proxyclasses=X,Y -proxyopts=multiple
```

Given these settings, JPublisher generates wrappers for instance methods, including Java wrapper classes and a PL/SQL wrapper. In this example, JPublisher generates the wrapper classes `XJPub` and `YJPub` for `X` and `Y`, respectively. Because of the `multiple` setting, JPublisher uses the handle mechanism to expose instance methods as static methods to allow access from PL/SQL (which supports only static methods). See "Mechanisms Used in Exposing Java to PL/SQL" on page 5-36 and "Wrapper Class Generation with Handles" on page 5-47 for related information.

Wrapper Class XJPub

Here is the class `XJPub`:

```
import java.util.Hashtable;

 public class XJPub
 {
```

```
public static void add(long handleX, int arg)
{ XJPub._getX(handleX).add(arg); }

public static void add(long handleX, long handleY)
{ XJPub._getX(handleX).add(YJPub._getY(handleY)); }

public static int getTot(long handleX)
{ return XJPub._getX(handleX).getTot(); }

public static long cloneTot(long handleX)
{ return YJPub._setY(XJPub._getX(handleX).cloneTot()); }

// Generic Code
//
// This code is generated similarly for all classes.
//

private static long m_seq;  // instance count
private static long m_base; // base represents null instance
private static Hashtable m_objs = new Hashtable();
private static Hashtable m_tags = new Hashtable();
static { m_base = XJPub.class.hashCode();
         m_base = (m_base>0l) ? -m_base : m_base;
         /* m_base = m_base && 0xFFFFFFFF00000000l */ ;
         m_seq = m_base;
       }

public static X _getX(long handleX)
{
  if (handleX<=m_base || handleX>m_seq) return null;
  return (X)m_objs.get(new Long(handleX));
}

public static long _setX(X x)
{
  if (x==null)
  {
    return m_base;
  }
  Long l = (Long)m_tags.get(x);

  if (l != null)
  {
    return l.longValue();
  }
  else
  {
    return newX();
  }
}

public static long newX()
{ m_seq++;
  Long l = new Long(m_seq);
  X x = new X();
  m_objs.put(l,x);
  m_tags.put(x,l);
  return m_seq;
}
```

```
  public static void releaseX(long handleX)
  { if (handleX<=m_base || handleX>m_seq)
    { /* wrong type - should we issue an error? */
      return;
    }
    Long l = new Long(handleX);
    m_objs.remove(l);
    Object x = m_objs.get(l);
    if (x!=null) m_tags.remove(x);
  }

  public static void releaseAllX()
  { m_objs.clear(); m_tags.clear(); m_seq=m_base; }
}
```

Wrapper Class YJPub

Here is the class `YJPub`:

```
import java.util.Hashtable;

 public class YJPub
 {
   public static void setTotal(long handleY, int total)
   { YJPub._getY(handleY).setTotal(total); }

   public static int getTotal(long handleY)
   { return YJPub._getY(handleY).getTotal(); }

   // Generic Code
   //
   // This code is generated similarly for all classes.
   //

   private static long m_seq;  // instance count
   private static long m_base; // base represents null instance
   private static Hashtable m_objs = new Hashtable();
   private static Hashtable m_tags = new Hashtable();
   static { m_base = YJPub.class.hashCode();
            m_base = (m_base>0l) ? -m_base : m_base;
            /* m_base = m_base && 0yFFFFFFFF00000000l */ ;
            m_seq = m_base;
          }

   public static Y _getY(long handleY)
   {
     if (handleY<=m_base || handleY>m_seq) return null;
     return (Y)m_objs.get(new Long(handleY));
   }

   public static long _setY(Y y)
   {
     if (y==null)
     {
       return m_base;
     }

     Long l = (Long)m_tags.get(y);

     if (l != null)
     {
       return l.longValue();
     }
     else
     {
       return newY();
     }
   }

   public static long newY()
   { m_seq++;
     Long l = new Long(m_seq);
     Y y = new Y();
     m_objs.put(l,y);
     m_tags.put(y,l);
     return m_seq;
   }

   public static void releaseY(long handleY)
   { if (handleY<=m_base || handleY>m_seq)
     { /* wrong type - should we issue an error? */
       return;
     }
     Long l = new Long(handleY);
     m_objs.remove(l);
     Object y = m_objs.get(l);
     if (y!=null) m_tags.remove(y);
   }

   public static void releaseAllY()
   { m_objs.clear(); m_tags.clear(); m_seq=m_base; }
 }
```

PL/SQL Wrapper

Here is the PL/SQL wrapper:

```
CREATE OR REPLACE PACKAGE JPUB_PLSQL_WRAPPER AS
    FUNCTION NEW_X() RETURN NUMBER;
    PROCEDURE ADD(handle_X NUMBER, arg0 NUMBER);
    PROCEDURE ADD2(handle_X NUMBER, handle_Y NUMBER);
    FUNCTION  GET_TOT(handle_X NUMBER) RETURN NUMBER;
    FUNCTION  CLONE_TOT(handle_X NUMBER) RETURN NUMBER;

    FUNCTION  NEW_Y() RETURN NUMBER;
    PROCEDURE SET_TOTAL(handle_Y NUMBER, arg0 NUMBER);
    FUNCTION  GET_TOTAL(handle_Y NUMBER) RETURN NUMBER;
 END JPUB_PLSQL_WRAPPER;
 /
 CREATE OR REPLACE PACKAGE BODY JPUB_PLSQL_WRAPPER AS
   FUNCTION NEW_X() RETURN NUMBER AS LANGUAGE JAVA
    NAME 'XJPub.newX() return long';

   PROCEDURE RELEASE_X(handle_X NUMBER) AS LANGUAGE JAVA
    NAME 'XJPub.releaseX(long)';

   PROCEDURE RELEASE_ALL_X() AS LANGUAGE JAVA
    NAME 'XJPub.releaseAllX()';

   PROCEDURE ADD(handle_X NUMBER, arg0 NUMBER) AS LANGUAGE JAVA
    NAME 'XJPub.add(long,int)';
```

```
  PROCEDURE ADD2(handle_X NUMBER, handle_Y NUMBER) AS LANGUAGE JAVA
   NAME 'XJPub.add(long,long)';

  FUNCTION GET_TOT(handle_X NUMBER) RETURN NUMBER AS LANGUAGE JAVA
   NAME 'XJPub.getTot(long) return int';

  FUNCTION CLONE_TOT(handle_X NUMBER) RETURN NUMBER AS LANGUAGE JAVA
   NAME 'XJPub.cloneTot(long) return long';

  FUNCTION NEW_Y() RETURN NUMBER AS LANGUAGE JAVA
   NAME 'YJPub.newY() return long';

  PROCEDURE RELEASE_Y(handle_Y NUMBER) AS LANGUAGE JAVA
   NAME 'YJPub.releaseY(long)';

  PROCEDURE RELEASE_ALL_Y() AS LANGUAGE JAVA
   NAME 'YJPub.releaseAllY()';

  PROCEDURE SET_TOTAL(handle_Y NUMBER, arg0 NUMBER) AS LANGUAGE JAVA
   NAME 'YJPub.setTotal(long,int)';

  FUNCTION GET_TOTAL(handle_Y NUMBER, arg0 NUMBER) RETURN NUMBER AS LANGUAGE JAVA
   NAME 'YJPub.getTotal(long) return int';

END JPUB_PLSQL_WRAPPER;
```

Index