Oracle9*i*

JPublisher User's Guide

Release 1 (9.0.1)

June 2001

Part No. A90214-01

ORACLE®

Oracle9*i* JPublisher User's Guide, Release 1 (9.0.1)

Part No. A90214-01

Copyright © 1999, 2001 Oracle Corporation. All rights reserved.

Primary Authors: Brian Wright, Thomas Pfaeffle, P. Alan Thiesen

Contributing Author: Janice Nygard

Contributors: Ekkehard Rohwedder, Prabha Krishna, Ellen Barnes

Contents

3 JPublisher Examples

Send Us Your Comments

Oracle9*i* JPublisher User's Guide, Release 1 (9.0.1)

Part No. A90214-01

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

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

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

- Electronic mail: jpgcomment_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 (optionally) electronic mail address.

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

Preface

This user's guide describes the Oracle JPublisher utility, which translates user-defined SQL object types and PL/SQL packages to Java classes. SQLJ and JDBC programmers who need to have Java classes in their applications to correspond to object types, VARRAY types, nested table types, object reference types, or PL/SQL packages can use the JPublisher utility.

Intended Audience

This manual assumes that you are an experienced Java programmer with knowledge of Oracle databases, SQL, PL/SQL, JDBC, and SQLJ.

Structure

This manual contains the following chapters:

- Chapter 1, "Understanding and Using JPublisher"—Introduces the JPublisher utility and provides an overview of its operations, including details regarding datatype mappings, generation of output classes, and support for inheritance.

- Chapter 2, "Command-Line Options and Input Files"—Describes the JPublisher command line, command line options, and input files.

- Chapter 3, "JPublisher Examples"—Presents examples of JPublisher output for object types and wrapper methods.

Related Documents

See the following additional documents available from the Oracle Java Platform group:

- *Oracle9i JDBC Developer's Guide and Reference*

  This book covers programming syntax and features of Oracle's implementation of the JDBC standard (for Java Database Connectivity). This includes an overview of the Oracle JDBC drivers, details of Oracle's implementation of JDBC 1.22 and 2.0 features, and discussion of Oracle JDBC type extensions and performance extensions.

- *Oracle9i SQLJ Developer's Guide and Reference*

  This book covers the use of SQLJ to embed static SQL operations directly into Java code, covering SQLJ language syntax and SQLJ translator options and features. Both standard SQLJ features and Oracle-specific SQLJ features are described.

- *Oracle9i Java Developer's Guide*

  This book introduces the basic concepts of Java in Oracle9*i* and provides general information about server-side configuration and functionality. Information that pertains to the Oracle Java platform as a whole, rather than to a particular product (such as JDBC, SQLJ, or EJBs), is in this book.

- *Oracle9i Servlet Engine Developer's Guide*

  This book documents use of the Oracle9*i* Servlet Engine, the servlet container in Oracle9*i*.

- *Oracle JavaServer Pages Developer's Guide and Reference*

  This book covers the use of JavaServer Pages technology to embed Java code and JavaBean invocations inside HTML pages. Both standard JSP features and Oracle-specific features are described.

- *Oracle9i Java Tools Reference*

  This book documents Java-related tools and utilities for use with Oracle9*i* or in deploying applications to Oracle9*i* (such as the session shell and `loadjava` tools).

- *Oracle9i Java Stored Procedures Developer's Guide*

  This book discusses Java stored procedures—programs that run directly in Oracle9*i*. With stored procedures (functions, procedures, triggers, and SQL

methods), Java developers can implement business logic at the server level, thereby improving application performance, scalability, and security.

- *Oracle9i Enterprise JavaBeans Developer's Guide and Reference*

  This book describes Oracle's Enterprise JavaBeans implementation and extensions.

- *Oracle9i CORBA Developer's Guide and Reference*

  This book describes Oracle's CORBA implementation and extensions.

The following documents from the Oracle Server Technologies group also contain information of interest and are referred to.

- *Oracle9i SQL Reference*

  This reference book contains a complete description of the content and syntax of the Structured Query Language (SQL) used to manage information in an Oracle database.

- *PL/SQL User's Guide and Reference*

  PL/SQL is Oracle's procedural extension to SQL. An advanced fourth-generation programming language, PL/SQL offers seamless SQL access, tight integration with the Oracle server and tools, portability, security, and features such as data encapsulation, overloading, exception handling, and information hiding. This guide explains the concepts behind PL/SQL and illustrates facets of the language.

Information about JDBC is also available at the following Sun Microsystems Web site:

```
http://java.sun.com/products/jdk/1.2/docs/guide/jdbc/
```

In particular, refer to the following Sun Microsystems specifications:

- *JDBC 2.0 API*
- *JDBC 2.0 Standard Extension API*

Conventions

This book uses Solaris syntax. However, file names and directory names for Windows NT are the same, unless otherwise noted.

The term [ORACLE_HOME] indicates the full path of the Oracle home directory.

Additionally, this manual uses the following conventions.

Convention	Meaning
italicized regular text	Italicized regular text is used for emphasis or to indicate a term that is being defined or will be defined shortly.
...	Horizontal ellipsis points in sample code indicate the omission of a statement or statements or part of a statement. This is done when you would normally expect additional statements or code to appear, but such statements or code would not be related to the example.
code text	Code text within regular text indicates class names, object names, method names, variable names, Java types, Oracle datatypes, file names, and directory names.
italicized_code_text	Italicized code text in a program statement indicates something that must be provided by the user.
<italicized_code_text>	Angle brackets enclosing italicized code text in a program statement indicates something that can *optionally* be provided by the user.

For more information, see "Notational Conventions" on page 2-5.

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. 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/

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.

1

Understanding and Using JPublisher

This chapter provides an overview of JPublisher followed by more detailed discussion of its operation. The following topics are covered:

- Understanding JPublisher
- JPublisher Operation
- Backwards Compatibility and Migration
- Details of Datatype Mapping
- Concepts of JPublisher-Generated Classes
- JPublisher Generation of SQLJ Classes
- JPublisher Generation of Java Classes
- User-Written Subclasses of JPublisher-Generated Classes
- JPublisher Support for Inheritance
- JPublisher Limitations

Understanding JPublisher

This section provides a basic understanding of what JPublisher is for and what it accomplishes, covering the following topics:

- Introduction to JPublisher

- What JPublisher Does

- What JPublisher Produces

- JPublisher Requirements

- JPublisher Input and Output

- Overview of Datatype Mappings

- Creating Types and Packages in the Database

Introduction to JPublisher

JPublisher is a utility, written entirely in Java, that generates Java classes to represent the following user-defined database entities in your Java program:

- SQL object types

- object reference types ("REF types")

- SQL collection types (VARRAY types or nested table types)

- PL/SQL packages

JPublisher enables you to specify and customize the mapping of SQL object types, object reference types, and collection types (VARRAYs or nested tables) to Java classes in a strongly typed paradigm.

JPublisher generates `getXXX()` and `setXXX()` accessor methods for each attribute of an object type. If your object types have stored procedures, JPublisher can generate wrapper methods to invoke the stored procedures. A wrapper method is a method that invokes a stored procedure that executes in Oracle9*i*.

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

The wrapper methods JPublisher generates contain SQLJ code, so when JPublisher generates wrapper methods, it generally produces `.sqlj` source files. This is true for classes representing PL/SQL packages or object types that define methods, unless you specify (through the `-methods` option) that JPublisher should *not* generate wrapper methods.

If no wrapper methods are generated, JPublisher produces `.java` source files. This is true for classes representing object types without methods, object reference types, or collection types, or for classes where the `-methods` option is off.

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

- Extend the generated classes.

- Write your own Java classes by hand, without using JPublisher. This approach is quite flexible, but time-consuming and error-prone.

- Use generic classes to represent object, object reference, and collection types. The `oracle.sql` package contains simple generic classes that represent object, object reference, and collection types. If these classes meet your requirements, you do not need JPublisher.

Object Types and JPublisher

JPublisher allows your Java language applications to employ user-defined object types in Oracle9*i*. If you intend to have your Java-language application access object data, then it must represent the data in a Java format. JPublisher helps you do this by creating the mapping between object types and Java classes, and between object attribute types and their corresponding Java types.

Classes generated by JPublisher implement either the `oracle.sql.ORAData` interface or the `java.sql.SQLData` interface, depending on how you set the JPublisher options. Either interface makes it possible to transfer object type instances between the database (or middle-tier database cache) and your Java program. For more information about the `ORAData` and `SQLData` interfaces, see the *Oracle9i JDBC Developer's Guide and Reference*.

PL/SQL Packages and JPublisher

You might want to call stored procedures in a PL/SQL package from your Java application. The stored procedure can be a PL/SQL subprogram or a Java method that has been published to SQL. Java arguments and functions are passed to and returned from the stored procedure.

To help you do this, you can direct JPublisher to create a class containing a wrapper method for each subprogram in the package. The wrapper methods generated by JPublisher provide a convenient way to invoke PL/SQL stored procedures from Java code or to invoke a Java stored procedure from a client Java program.

If you call PL/SQL code that includes top-level subprograms (subprograms not in any PL/SQL package), JPublisher generates a single class containing wrapper methods for the top-level subprograms you request.

What JPublisher Does

JPublisher connects to a database and retrieves descriptions of the SQL object types or PL/SQL packages that you specify on the command line or from an input file. By default, JPublisher connects to the database by using the JDBC OCI driver, which requires an Oracle client installation, including Oracle Net and required support files. If you do not have an Oracle client installation, JPublisher can use the Oracle JDBC Thin driver.

JPublisher generates a Java class for each SQL object type it translates. The Java class includes 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 you create wrapper methods (Java methods to wrap stored procedures or functions of the SQL object type), JPublisher generates SQLJ source code so you must additionally have the SQLJ runtime libraries.

When you call a wrapper method, the SQL value for the object is sent to the server, along with any `IN` our `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.

JPublisher also generates a class for each PL/SQL package it translates. The class includes code to invoke the package methods 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.

The next section furnishes a general description of the source files that JPublisher creates for object types and PL/SQL packages.

What JPublisher Produces

The number of files JPublisher produces depends on whether you request `ORAData` classes (classes that implement the `oracle.sql.ORAData` interface) or `SQLData` classes (classes that implement the standard `java.sql.SQLData` interface).

The `ORAData` interface supports SQL object, object reference, and collection types in a strongly typed way. That is, for each specific object, object reference, or collection type in the database, there is a corresponding Java type. The `SQLData` interface, on the other hand, supports only SQL object types in a strongly typed way. All object

reference types are represented generically as `java.sql.Ref` instances, and all collection types are represented generically as `java.sql.Array` instances. Therefore, JPublisher generates classes for object reference and collection types only if it is generating `ORAData` classes.

When you run JPublisher for a user-defined object type and you request `ORAData` classes, JPublisher automatically creates the following:

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

- a related reference class for object references to your Oracle object type

- Java classes for any object or collection attributes nested directly or indirectly within the top-level object (necessary so that attributes can be materialized in Java whenever an instance of the top-level class is materialized)

> **Note:** 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 "Strongly Typed Object References for ORAData Implementations" on page 1-14.

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

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

- a collection class to act as a type definition to correspond to your Oracle collection type

- 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 (necessary so object elements can be materialized in Java whenever an instance of the collection is materialized)

When you run JPublisher for a PL/SQL package, it automatically creates the following:

- a Java class with wrapper methods that invoke the stored procedures of the package

JPublisher Requirements

JPublisher requires that Oracle SQLJ and Oracle JDBC also be installed on your system and in your classpath appropriately. You will need the following libraries (all available as either `.zip` or `.jar`):

- SQLJ translator classes (`translator`)
- SQLJ runtime classes (`runtime12`, `runtime12ee`, `runtime11`, `runtime`, or `runtime-nonoracle`)
- JDBC classes (`classes12` or `classes111`)

"12" refers to versions for JDK 1.2.x; "11" and "111" refer to versions for JDK 1.1.x. See the *Oracle9i SQLJ Developer's Guide and Reference* for more information about these files.

When you use Oracle9*i* JPublisher, you should also use the equivalent version of SQLJ, because these two products are always installed together. To use all features of JPublisher, you also need:

- Oracle9*i* (or version 8.1.7 or 8.1.6)
- Oracle9*i* JDBC drivers (or version 8.1.7 or 8.1.6)
- Java Developer's Kit (JDK) version 1.2

If you are using only some features of JPublisher, your requirements might be less stringent:

- If you never generate `SQLData` classes, and you never use the `java.sql.Blob` and `java.sql.Clob` classes, you can use JDK version 1.1.x instead of JDK 1.2.x.
- If you never generate code for PL/SQL packages, you can use Oracle database version 8.1.5.
- If you never generate classes that implement the Oracle-specific `ORAData` interface (or the deprecated `CustomDatum` interface), you should be able to use a non-Oracle JDBC driver or a non-Oracle SQLJ implementation. When running code generated by JPublisher, you should even be able to connect to a non-Oracle database; however, JPublisher itself must connect to an Oracle database. Oracle does not test or support configurations that use non-Oracle components.
- If you never use PL/SQL packages or classes that implement the `SQLData` interface (that is, you use JPublisher to generate only classes that implement the `ORAData` interface or the deprecated `CustomDatum` interface), you can use

Oracle database version 8.1.5 with JDBC version 8.1.5 and JDK version 1.1.x or higher. (Be aware that the ORAData interface requires the Oracle9*i* or higher JDBC implementation.)

■ 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 files. In this case, you would not need the SQLJ translator. See "Generate Classes for Packages and Wrapper Methods for Methods (-methods)" on page 2-17 for information about the -methods option.

JPublisher Input and Output

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

JPublisher Input

You can specify JPublisher options on the command line or in a properties file. "JPublisher Options" on page 2-2 describes all the JPublisher options.

In addition, you can use a file known as the INPUT file to specify the object types and PL/SQL packages JPublisher should translate. It also controls the naming of the generated packages and classes. "INPUT File Structure and Syntax" on page 2-26 describes INPUT file syntax.

A properties file is an optional text file that you can use to specify frequently-used options. You name the properties file on the command line. JPublisher processes the properties file as if its contents were inserted, in sequence, on the command line at that point. For more information about this file, see "Properties File Structure and Syntax" on page 2-25.

JPublisher Output

JPublisher generates a Java class for each object type that it translates. For each object type, JPublisher generates a <*type*>.sqlj file (or a <*type*>.java file if wrapper methods were suppressed or do not exist) for the class code and a <*type*>Ref.java file for the code for the REF class of the Java type. For example, if you define an EMPLOYEE SQL object type, JPublisher generates an employee.sqlj file (or an employee.java file) and an employeeRef.java file. Note that the case of Java class names produced by JPublisher is determined by the -case option. See "Case of Java Identifiers (-case)" on page 2-12.

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.

For PL/SQL packages, JPublisher generates classes containing wrapper methods as `.sqlj` files.

When JPublisher generates the class files and wrappers, it also writes the names of the translated types and packages to standard output.

Overview of Datatype Mappings

JPublisher offers different categories of datatype mappings from SQL to Java. JPublisher options to specify these mappings are described below, under "Detailed Descriptions—Options That Affect Datatype Mappings" on page 2-6.

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 "Details of Datatype Mapping" on page 1-26.

JDBC Mapping

The JDBC mapping maps most numeric datatypes to Java primitive types such as `int` and `float`, and maps DECIMAL and NUMBER to `java.math.BigDecimal`. LOB types and other non-numeric built-in types map to standard JDBC Java types such as `java.sql.Blob` and `java.sql.Timestamp`. For object types, JPublisher generates SQLData classes. Predefined datatypes that are Oracle extensions (such as BFILE and ROWID) do not have JDBC mappings, so 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`, 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

The Object JDBC mapping maps most numeric datatypes to Java wrapper classes such as `java.lang.Integer` and `java.lang.Float`, and maps `DECIMAL` and `NUMBER` to `java.math.BigDecimal`. It 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`, 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`, an exception is thrown if the value is too large to fit.

This is the default mapping for numeric types.

BigDecimal Mapping

BigDecimal mapping, as the name implies, maps all numeric datatypes to `java.math.BigDecimal`. It supports null values and very large values.

Oracle Mapping

In the Oracle mapping, JPublisher maps any numeric, LOB, or other built-in type to a class 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.

Creating Types and Packages in the Database

Before you run JPublisher, you must create any new datatypes that you will require in the database. You must also ensure that any PL/SQL packages, methods, and subprograms that you want to invoke from Java are also installed in Oracle9*i*.

Use the SQL `CREATE TYPE` statement to create object, VARRAY, and nested table types in the database. JPublisher supports the mapping of these datatypes to Java classes. JPublisher also generates classes for references to object types. REF types are

not explicitly declared in SQL. For more information on creating object types, see the *Oracle9i SQL Reference.*

Use the CREATE PACKAGE and CREATE PACKAGE BODY statements to create PL/SQL packages and store them in the database. PL/SQL furnishes all the capabilities necessary to implement the methods associated with object types. These methods (functions and procedures) reside on the server as part of a user's schema. You can implement the methods in PL/SQL or Java.

Packages are often implemented to provide the following advantages:

- encapsulation of related procedures and variables
- declaration of public and private procedures, variables, constants, and cursors
- better performance

For more information on PL/SQL and creating PL/SQL packages, see the *PL/SQL User's Guide and Reference.*

JPublisher Operation

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

- Translating and Using PL/SQL Packages and User-Defined Types
- Representing User-Defined Object, Collection, and Reference Types in Java
- Strongly Typed Object References for ORAData Implementations
- JPublisher Command-Line Syntax
- Sample JPublisher Translation

Translating and Using PL/SQL Packages and User-Defined Types

Here are the basic steps for translating and using code for user-defined types and PL/SQL packages. (User-defined types include Oracle objects and Oracle collections—VARRAYs and nested tables.)

1. Create the desired user-defined datatypes and PL/SQL packages in the database.

2. JPublisher generates source code for Java classes that represent PL/SQL packages, user-defined types, and reference types and places them in specified Java packages. JPublisher generates `.java` files for object reference, VARRAY, and nested table classes. If you instruct JPublisher to generate wrapper methods, it will generate `.sqlj` files for packages and object types (assuming the object types have methods). If you instruct JPublisher to *not* generate wrapper methods, it will generate `.java` files without wrapper methods for object types and will not generate classes for packages (because they contain only wrapper methods). For object types without methods, JPublisher generates `.java` files in any case.

3. Import these classes into your application code.

4. Use the methods in the generated classes to access and manipulate the user-defined types and their attributes.

5. Compile all classes (the JPublisher-generated code and your code). SQLJ translates the `.sqlj` files, and the Java compiler compiles the `.java` files.

6. Run your compiled application.

Figure 1-1 illustrates the preceding steps.

Figure 1–1   Translating and Using Object Code
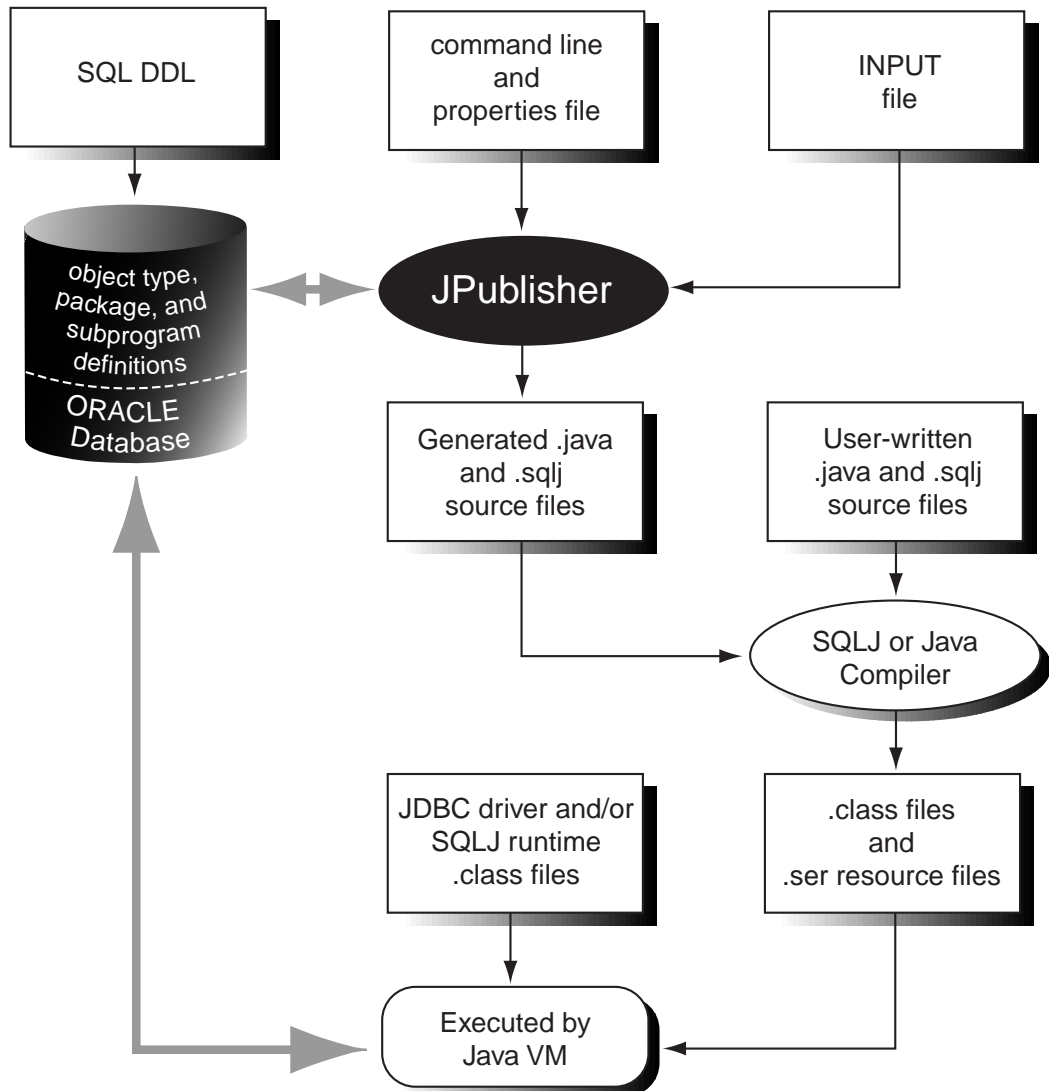
Representing User-Defined Object, Collection, and Reference Types in Java

Here are the three ways to represent user-defined object, collection, and object reference types in your Java program:

- Use classes that implement the ORAData interface.

  JPublisher generates classes that implement the ORAData interface. (You can also write them by hand, but this is not generally recommended.)

- Use classes that implement the SQLData interface, as described in the JDBC 2.0 API.

  JPublisher generates classes for SQL object types that implement the SQLData interface. (You can also write them by hand, but this is not generally recommended. Be aware 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 instances, and all collection types are represented generically as java.sql.Array instances.

- 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, and the class oracle.sql.REF represents all REF types. These classes are immutable in the same way that java.lang.String is.

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 a comparison of the SQLData and ORAData interfaces, see the *Oracle9i JDBC Developer's Guide and Reference*.

Compared to oracle.sql.* classes, classes that implement ORAData or SQLData are strongly typed. Your connected SQLJ translator will detect an error at translation time if, for example, you mistakenly select a PERSON object into an ORAData object that represents an ADDRESS.

JPublisher-generated classes that implement ORAData or SQLData have additional advantages:

- The classes are customized, rather than generic. You access attributes of an object using getXXX() and setXXX() methods named after the particular

attributes of the object. Note that you have to 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.

Strongly Typed Object References for ORAData Implementations

For Oracle ORAData implementations, JPublisher always generates strongly typed object reference classes as opposed to using the weakly typed oracle.sql.REF class. This is to provide greater type safety and to mirror the behavior in SQL, where object references are strongly typed. The strongly typed classes (with names such as PersonRef for references to PERSON objects) are essentially wrappers for the REF class.

In these strongly typed REF wrappers, there is a getValue() method that 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.

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.

For standard SQLData implementations, strongly typed object references are not supported—they are not part of the standard. JPublisher does not create a custom reference class; you must use java.sql.Ref or oracle.sql.REF as the reference type.

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 files and writing the names of the translated object types or PL/SQL packages to standard output.

Here is an example of a command that invokes JPublisher (this is a single wrap-around command line):

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

You enter the command on one line, allowing it to wrap as necessary. For clarity, this chapter refers to the input file (the 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 username SCOTT and password TIGER and 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 in the directory demo.

"JPublisher Options" on page 2-2 describes each of these options in more detail.

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

Sample JPublisher Translation

This section provides a sample JPublisher translation of an 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.

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 choice for the value of the `-numbertypes` option.

JPublisher translates the types according to the following command line:

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

(This is a single wrap-around command line.)

"JPublisher Options" on page 2-2 describes each of these options in detail.

Note that because the `EMPLOYEE` object type does not define any methods, JPublisher will generate a `.java` file, not a `.sqlj` file.

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:

```
./demo/corp/Employee.java          (UNIX)
.\demo\corp\Employee.java          (Windows NT)
```

The `Employee.java` class file would contain the code below.

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;

  static int[] _sqlType =
  {
    12, 4, 2, 91, 7
  };

  static ORADataFactory[] _factory = new ORADataFactory[5];

  static final Employee _EmployeeFactory = new Employee();
  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); }

  /* 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

- For Oracle9*i* releases (as well as Oracle8*i* release 8.1.7), there is a protected _struct field in JPublisher-generated code for SQL object types. This is an instance of the internal class MutableStruct; this instance contains the data in original SQL format. In general, you should not reference this field directly. Instead, use the setting -methods=always or -methods=named as necessary to ensure that JPublisher produces .sqlj files, then use the methods setFrom() and setValueFrom() when subclassing. See "The setFrom() and setValueFrom() Methods" on page 1-48.

- In Oracle8*i* compatibility mode, there is also a protected _ctx field that is a SQLJ connection context instance. See "Oracle8i Compatibility Mode" on page 1-24 for more information.

- Note that Oracle8*i* JPublisher would generate 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;

  static final EmployeeRef _EmployeeRefFactory = new EmployeeRef();
  public static ORADataFactory getORADataFactory()
  {
```

```
        return _EmployeeRefFactory;
      }

      /* constructor */
      public EmployeeRef()
      {
      }

      /* ORAData interface */
      public Datum toDatum(java.sql.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 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()));
      }
    }
}
```

You can find more examples of object mappings in "Example: JPublisher Object Attribute Mapping" on page 3-8.

Backwards Compatibility and Migration

This section discusses issues of backwards compatibility, compatibility between JDK versions, and migration between Oracle8*i* and Oracle9*i* releases of JPublisher.

JPublisher Backwards Compatibility

The JPublisher runtime is packaged with Oracle JDBC in the `classes111` or `classes12` library. Code generated by an earlier version of JPublisher will:

- continue to run with the current release of the JPublisher runtime
- continue to be compilable against the current release of the JPublisher runtime

If you use an earlier release of the JPublisher runtime and Oracle JDBC in generating code, the code will be compilable against that version of the JPublisher runtime. Specifically, when you use an Oracle8*i* JDBC driver, JPublisher will generate code for the now-deprecated `CustomDatum` interface, not the `ORAData` interface that replaced it.

JPublisher Compatibility Between JDK Versions

Generally speaking, `.sqlj` files generated by JPublisher can be translated under either JDK 1.1.x (assuming you are not using JDBC 2.0-specific types) or JDK 1.2.x or higher. However, if you intend to translate and compile in separate steps (setting `-compile=false` in SQLJ so that only `.java` files, not `.class` files, are produced), then you must use the same JDK version for compilation as for translation unless you use a special JPublisher option setting.

In this situation (translating and compiling in separate steps), the JPublisher default setting `-context=DefaultContext` results in generation of `.sqlj` files that are completely compatible between JDK 1.1.x and JDK 1.2.x or higher. (With this setting, for example, you could translate against JDK 1.1.x but still compile against JDK 1.2.x successfully.)

In this situation, all generated `.sqlj` files use the `sqlj.runtime.ref.DefaultContext` class for all connection contexts. This is as opposed to the setting `-context=generated`, which results in each generated `.sqlj` file declaring its own connection context inner class. This was the Oracle8*i* JPublisher default behavior, and is what makes translated `.java` code incompatible between JDK 1.1.x and 1.2.x or higher.

See "SQLJ Connection Context Classes (-context)" on page 2-13 for more information about the `-context` option.

> **Important:** With some JPublisher option settings under JDK 1.1.x there is risk of memory leakage caused by SQLJ connection context instances that are not closed. See "Releasing Connection Context Resources" on page 1-40 for information.

See the *Oracle9i SQLJ Developer's Guide and Reference* for general information about connection contexts.

Migration Between Oracle8*i* JPublisher and Oracle9*i* JPublisher

In Oracle9*i* JPublisher, default option settings and some features of the generated code have changed. If you wrote an application using JPublisher release 8.1.7 or earlier, it is unlikely that you will be able to simply re-run JPublisher in Oracle9*i* and have the generated classes still work within your application. This section describes how to modify your JPublisher option settings or your application code appropriately.

> **Note:** Also see "Changes in User-Written Subclasses of Oracle9i JPublisher-Generated Classes" on page 1-46 for differences between Oracle8*i* functionality and Oracle9*i* functionality for classes that extend JPublisher-generated classes.

Changes in Behavior in Oracle9*i* JPublisher

Be aware of the following changes in JPublisher behavior 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.

  Also, user-written code must call the `getConnectionContext()` method to have a connection context handle, instead of using the `_ctx` connection context field that was declared under Oracle8*i* code generation. See "Considerations in Using Connection Contexts and Connection Instances" on page 1-38 for more information about the `getConnectionContext()` method.

- The setting `-methods=true` will result in `.java` files being generated instead of `.sqlj` files if the underlying SQL object type or PL/SQL package does not define any methods.

- By default, JPublisher now generates code that implements the `oracle.sql.ORAData` interface instead of the deprecated `oracle.sql.CustomDatum` interface.

- By default, JPublisher now simply places generated code into the current directory, rather than into a package-directory hierarchy under the current directory.

See the following sections, "Individual Settings to Force JPublisher Behavior as in Previous Releases" and "Oracle8i Compatibility Mode", for information about how to revert to Oracle8*i* behavior instead.

Individual Settings to Force JPublisher Behavior as in Previous Releases

In Oracle9*i*, if you want JPublisher to behave as it did in release 8.1.7 and prior, there are a number of individual backwards-compatibility options you can set. These are detailed in Table 1–1. See descriptions of these options under "Detailed Descriptions—General JPublisher Options" on page 2-11 for more information.

See "Oracle8i Compatibility Mode" on page 1-24 for a single setting that results in the same behavior as for Oracle8*i* JPublisher—backwards-compatible code generation plus behavior that is equivalent to what would happen with the combination of these individual option settings.

Table 1–1   JPublisher Backwards-Compatibility Options

Option Setting	Behavior
-context=generated	This 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 forces generation of `.sqlj` (as opposed to `.java`) source files for all JPublisher-generated classes, regardless of whether the underlying SQL object or package actually defines any methods.
-compatible=customdatum	For Oracle-specific object wrappers, this results in JPublisher implementing the older `oracle.sql.CustomDatum` and `CustomDatumFactory` interfaces (both deprecated but still supported in Oracle9*i*) instead of the newer `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*.

Unless you have a compelling reason to use the backwards-compatibility settings, however, it is recommended that you accept the current default (or other) settings.

Oracle8*i* Compatibility Mode

The following setting results in what is called *Oracle8i compatibility mode*:

```
-compatible=8i
```

See "Backwards-Compatible Oracle Mapping for User-Defined Types (-compatible)" on page 2-7 for more information about this option.

For use of this mode to be permissible, however, at least one of the following circumstances must hold:

- You will compile JPublisher-generated code with the SQLJ `-codegen=oracle` setting.

- The JPublisher-generated code will execute under JDK 1.2 or higher and will use the SQLJ `runtime12` or `runtime12ee` library, or it will execute in the Oracle9*i* release of the Oracle JVM.

- You run JPublisher with the `-methods=false` (or `none`) setting.

JPublisher has the following functionality in Oracle8*i* compatibility mode:

- JPublisher will generate code that implements the deprecated `CustomDatum` API instead of the `ORAData` API (as with `-compatible=customdatum`).

- With the default `-methods=true` setting, JPublisher will always generate SQLJ source code for a SQL object type, even if the object type does not define any methods (as with `-methods=always`).

- JPublisher will generate connection context declarations and connection context instances on every object type wrapper, 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 Oracle9*i*, 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, Oracle9*i* JPublisher 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.

  > **Important:** There are circumstances where you should not use Oracle8*i* compatibility mode. If your environment uses any of the following:
  >
  > - JDK 1.1.x, the SQLJ `runtime` library, or the SQLJ `runtime11` library
  >
  > and you use the following JPublisher setting:
  >
  > - `-codegen=iso`
  >
  > as well as any of the following settings:
  >
  > - `-methods=named` (or `some`) or `-methods=true` (or `all`)
  >
  > then there may be significant memory leakage caused by implicit connection context instances that are not closed.
  >
  > Avoid the `-compatible=8i` setting in these circumstances, and use the `setConnectionContext()` and `release()` methods in manipulating connection contexts. For more information, see "Use of Connection Contexts and Instances in SQLJ Code Generated by JPublisher" on page 1-37.

  > **Note:** Oracle8*i* compatibility mode is now the only way for a connection context instance `_ctx` to be declared in JPublisher-generated code (there is no other option setting to accomplish 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.

Details of Datatype Mapping

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

- jdbc

- objectjdbc (for -numbertypes only)

- bigdecimal (for -numbertypes only)

- oracle

These mappings, described in "Overview of Datatype Mappings" on page 1-8, affect the argument and result types JPublisher uses in the methods it generates.

The class that JPublisher generates for an object type will have getXXX() and setXXX() methods for the object attributes. The class that JPublisher generates for a VARRAY or nested table type will have getXXX() and setXXX() methods that access the elements of the array or nested table. When you use the option -methods=true, the class that JPublisher generates for an object type or PL/SQL package will have wrapper methods that invoke server methods of the object type or package. The mapping options control the argument and result types these methods will use.

The JDBC and Object JDBC mappings use familiar Java types that can be manipulated using standard Java operations. If your JDBC program is manipulating Java objects stored as object types, you might prefer the JDBC or Object JDBC mapping.

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. 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.

When you decide which mapping to use, you should remember that data format conversion is only a part of the cost of transferring data between your program and the server.

Datatype Mapping Tables

Table 1–2 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 1-28.

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

The **Oracle Mapping** column lists the corresponding Java types 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.

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. Refer to the *Oracle9i JDBC Developer's Guide and Reference* for more information on the oracle.sql package.

A few datatypes are not currently supported by JPublisher, in some cases because they are not directly supported by Oracle SQLJ or JDBC, as noted in the table.

Table 1–2 PL/SQL Datatype to Oracle and Object 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	*not currently supported by JPublisher*	*not currently supported by JPublisher*
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	short
REAL	oracle.sql.NUMBER	float

Table 1–2   PL/SQL Datatype to Oracle and Object JDBC Mapping Classes (Cont.)

SQL and PL/SQL Datatype	Oracle Mapping	JDBC Mapping
DATE	oracle.sql.DATE	java.sql.Timestamp
ROWID, UROWID	oracle.sql.ROWID	oracle.sql.ROWID
BOOLEAN	*not directly supported by JDBC*	*not directly supported by JDBC*
CLOB	oracle.sql.CLOB	java.sql.Clob
BLOB	oracle.sql.BLOB	java.sql.Blob
BFILE	oracle.sql.BFILE	oracle.sql.BFILE
NCLOB	*not currently supported by JPublisher*	*not currently supported by JPublisher*
object types	generated class	generated class
SQLJ object types	*not directly supported by SQLJ*	*not directly supported by SQLJ*
OPAQUE types	*not directly supported by SQLJ*	*not directly supported by SQLJ*
RECORD types	*not directly supported by JDBC*	*not directly supported by JDBC*
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
index-by tables	*not directly supported by SQLJ*	*not directly supported by SQLJ*
user-defined subtypes	same as for base type	same as for base type

The Object JDBC and BigDecimal mappings, which affect numeric types only, are fully described in "Mappings For Numeric Types (-numbertypes)" on page 2-8.

Allowed Object Attribute Types

You can use a subset of the PL/SQL datatypes listed in Table 1–2 as object attribute types. These datatypes are listed here and have the same Oracle mappings and JDBC mappings as described in the table:

- CHAR, VARCHAR, VARCHAR2, CHARACTER

- DATE

- DECIMAL, DEC, NUMBER, NUMERIC

- `DOUBLE PRECISION, FLOAT`

- `INTEGER, SMALLINT, INT`

- `REAL`

- `RAW, LONG RAW`

- `CLOB`

- `BLOB`

- `BFILE`

- object type

- nested table, VARRAY type

- reference type

Using Datatypes Not Supported by JDBC

JPublisher cannot generate wrapper methods for PL/SQL methods that use datatypes not directly supported by JDBC. If you must call a PL/SQL method that uses unsupported datatypes (such as `BOOLEAN`), you have two choices:

- Rewrite the PL/SQL method to avoid using the type.

- Write an anonymous block that converts input types that JDBC supports into the input types used by the PL/SQL method, and converts output types used by the PL/SQL method into output types that JDBC supports. For more information on this technique, see "Example: Using Datatypes Not Supported by JDBC" on page 3-70.

Concepts of JPublisher-Generated Classes

This section covers basic concepts about the code that JPublisher produces, including the following:

- how output parameters of SQL object type methods and PL/SQL methods are treated

- how member methods are called

- how overloaded methods are handled

For more information, see the following sections later in this chapter:

- JPublisher Generation of SQLJ Classes

- JPublisher Generation of Java Classes

- JPublisher Support for Inheritance

Passing OUT Parameters

Stored procedures called through SQLJ do not have the same parameter-passing behavior as ordinary Java methods. This affects the code you write when you call a wrapper method 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.

In contrast, when you call a stored procedure through SQLJ, 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 parameter that has been modified appear in separate objects.

A wrapper method JPublisher generates contains SQLJ code to call a 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, and IN OUT. The IN OUT and OUT parameters of the stored procedure 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.

Passing Parameters Other Than the "this" Parameter

The simplest way to solve the problem described above is to pass an OUT or IN OUT parameter to the wrapper method in a single-element array. The array is a sort of container that holds the parameter.

- 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.

In the following example, you have an initialized variable p of class Person, and x is an object belonging to a JPublisher-generated class that has a wrapper method f taking an IN OUT Person argument. You create the array and pass the parameter as follows:

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

Unfortunately, this technique for passing OUT or IN OUT parameters requires you to add a few extra lines of code to your program for each parameter. If your stored program has many OUT or IN OUT parameters, you might prefer to call it directly using SQLJ code, rather than a wrapper method.

Passing the "this" Parameter

Problems similar to what is described above arise when the this object of an instance method is modified.

The this object is an additional parameter that is passed in a different way. Its mode, as declared in the CREATE TYPE statement, may be IN or IN OUT. If you do not explicitly declare the mode of 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, the wrapper method must return the new value of this. The code generated by JPublisher processes this 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 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 would define the following method:

```
public 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 the SQL object type `MYTYPE` has the following member function:

```
MEMBER FUNCTION f2(x IN INTEGER) RETURNS VARCHAR2;
```

Then the corresponding Java class `MyJavaType` would define the following method:

```
public 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 (a SQLJ connection context instance or JDBC connection instance) with each wrapper class instance. The connection instance is used in initializing the wrapper class instance, so that subsequently you are not required to explicitly provide a connection or connection context instance when calling wrapper methods.

Translating Overloaded Methods

PL/SQL, as with Java, lets you create overloaded methods—two or more methods with the same name, but different signatures. If you use JPublisher to generate wrapper methods for PL/SQL methods, it is possible that two overloaded methods with different signatures in PL/SQL might have identical signatures in Java. If this occurs, JPublisher changes the names of the methods to avoid generating two or more methods with the identical signature. For example, consider a PL/SQL package or object type that includes these functions:

```
FUNCTION f(x INTEGER, y INTEGER) RETURN INTEGER
```

and

```
FUNCTION f(xx FLOAT, yy FLOAT) RETURN INTEGER
```

In PL/SQL, these functions have different argument types. However, once they are translated to Java with Oracle mapping, this difference disappears (both INTEGER and FLOAT map to oracle.sql.NUMBER).

Suppose that JPublisher generates a class for the package or object type with the command-line setting -methods=true and Oracle mapping. JPublisher responds by generating code similar to this:

```
public oracle.sql.NUMBER f_1 (
  oracle.sql.NUMBER x,
  oracle.sql.NUMBER y)
throws SQLException
{
  /* body omitted */
}

public oracle.sql.NUMBER f_4 (
  oracle.sql.NUMBER xx,
  oracle.sql.NUMBER yy)
throws SQLException
{
  /* body omitted */
}
```

Note that in this example, JPublisher names the first function f_1 and the second function f_4. Each function name ends with _<nn>, where <nn> is a number assigned by JPublisher. The number has no significance of its own, but JPublisher uses it to guarantee that the names of functions with identical parameter types will be unique.

JPublisher Generation of SQLJ Classes

When `-methods=true`, JPublisher generates `.sqlj` files for PL/SQL packages and for object types (unless an object type does not define any methods, in which case a `.java` file is generated). The classes includes wrapper methods that invoke the server methods of the object types and packages. Run SQLJ to translate the `.sqlj` file.

This section describes how to use these generated classes in your SQLJ code.

Use of SQLJ Classes JPublisher Generates for PL/SQL Packages

To use a class that JPublisher generates for a PL/SQL package:

- Construct an instance of the class.
- 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, and 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. Oracle JDBC provides the constructor that takes a `Connection` instance for the convenience of the JDBC programmer who knows how to compile a SQLJ program, but is unfamiliar with SQLJ concepts such as `DefaultContext`.

> **Important:** Classes produced by JPublisher include a `release()` method. In creating and using an instance of a JPublisher-generated wrapper class, if you do not use the constructor with the `DefaultContext` argument, and you do not subsequently call the `setConnectionContext()` method with a connection context argument, and you invoke a wrapper method, then the wrapper object will implicitly construct a `DefaultContext` instance. In this case, use the `release()` method to release the connection context instance when it is no longer needed. See "Use of Connection Contexts and Instances in SQLJ Code Generated by JPublisher" on page 1-37 for more information.

> **Note:** In Oracle8*i* JPublisher and 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 (an instance of any class that implements the standard `sqlj.runtime.ConnectionContext` interface).

The wrapper methods are all instance methods, because the connection context in the `this` object is used in `#sql` statements 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 you use. If the default context is the only one you use, call the no-argument constructor once. The *Oracle9i SQLJ Developer's Guide and Reference* discusses reasons for using explicit connection context instances.

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.

"Example: Using Classes Generated for Packages" on page 3-65 shows how to use a class generated for a PL/SQL package.

Use of Classes JPublisher Generates for Object Types

To use an instance of a Java class that JPublisher generates for a SQL object type, you must first initialize the Java object.

To initialize your Java object, you can:

- Assign an already initialized Java object to your Java object.

- Retrieve a copy of a SQL object into your Java object. To do this, you can:

    - Use the SQL object as an `OUT` argument or as the function call return of a JPublisher-generated wrapper method.

    - Retrieve the SQL object through `#sql` statements you write.

    - Retrieve the SQL object through JDBC calls you write.

- Construct the Java object and set its attributes using the `setXXX()` methods.

    The constructors for the class associate a connection with the class instance. One constructor takes a `DefaultContext` instance (or an instance of a class

specified through the `-context` option when you run JPublisher), one constructor takes a `Connection` instance, and 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. Oracle JDBC provides the constructor that takes a `Connection` instance for the convenience of the JDBC programmer who knows how to compile a SQLJ program, but is unfamiliar with SQLJ concepts such as `DefaultContext`.

> **Important:** Classes produced by JPublisher include a `release()` method. In creating and using an instance of a JPublisher-generated wrapper class, if you do not use the constructor with the `DefaultContext` argument, and you do not subsequently call the `setConnectionContext()` method with a connection context argument, and you invoke a wrapper method, then the wrapper object will implicitly construct a `DefaultContext` instance. In this case, use the `release()` method to release the connection context instance when it is no longer needed. See "Use of Connection Contexts and Instances in SQLJ Code Generated by JPublisher" on page 1-37 for more information.

> **Note:** In Oracle8*i* JPublisher and 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 (an instance of any class that implements the standard `sqlj.runtime.ConnectionContext` interface).

Once you have initialized your Java object, you can:

- 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 `#sql` statements.
- Use the object as a host variable in JDBC calls.

There is a Java attribute for each attribute of the corresponding SQL object type. The object has `getXXX()` and `setXXX()` 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 executing in the server. The wrapper methods are all instance methods, regardless of whether the server methods are. The `DefaultContext` in the `this` object is used in `#sql` statements 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; however, you may want to use them if converting from one object reference wrapper type to another.

The `RationalO` example, described in "Example: Using Classes Generated for Object Types" on page 3-53, shows how to use a class that was generated for an object type and has wrapper methods.

Use of Connection Contexts and Instances in SQLJ Code Generated by JPublisher

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 setting) results in JPublisher using instances of the standard `sqlj.runtime.ref.DefaultContext` class.

- A setting of a user-specified class (that is in the classpath and implements the standard `sqlj.runtime.ConnectionContext` interface) results in JPublisher using instances of that class.

- A setting of `-context=generated` results in the following declaration in the JPublisher-generated class:

```
#sql static context _Ctx
```

In this case, JPublisher uses instances of the `_Ctx` class for connection context instances.

> **Note:** It is no longer routine (as it was in Oracle**8i** JPublisher) for JPublisher to declare a connection context instance _ctx. This only occurs in Oracle**8i** compatibility mode (-compatible=8i), 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 "Considerations in Using Connection Contexts and Connection Instances" below for more information about these methods.

See "SQLJ Connection Context Classes (-context)" on page 2-13 for more information about the -context option.

Considerations in Using Connection Contexts and Connection Instances

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 you can use to explicitly specify a SQLJ connection context instance. (This will not be necessary if you have already specified a connection context instance through the constructor.)

  This method is defined as follows:

  ```
  public void setConnectionContext(conn_ctxt_instance);
  ```

  This installs the passed connection context instance as the SQLJ connection context in the object wrapper instance. The connection context instance must be an instance of the class specified through the -context option for JPublisher connection contexts (typically DefaultContext).

  Be aware 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 only valid 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 only occurs with implicitly created connection context instances.

- Use either of the following methods of an object wrapper instance, as appropriate, to retrieve a connection or connection context instance.

    - `Connection getConnection()`

    - *Conn_Ctxt_Type* `getConnectionContext()`

    The `getConnectionContext()` method returns an instance of the connection context class specified through the JPublisher `-context` option (typically `DefaultContext`).

    The returned connection context instance might either be an instance that was set explicitly through the `setConnectionContext()` method, or an instance that was created implicitly by JPublisher.

    > **Note:** These methods are available only in generated `.sqlj` files, not generated `.java` files. If necessary, you can use the setting `-methods=always` to ensure that `.sqlj` files are produced.

- If code in a JPublisher-generated class uses any SQLJ statements, and you do not set a connection context instance explicitly, then one will be created implicitly from the JDBC connection instance when the `getConnectionContext()` method is called.

    In this circumstance, you must be careful to use the `release()` method to free resources in the SQLJ runtime that would otherwise result in a memory leak.

- Having different connection context classes in different generated classes gives you the option of checking different classes against different exemplar schemas during SQLJ on-line semantics checking; however, because the SQLJ source is constructed from actual SQL types, this checking is usually not necessary.

See "Releasing Connection Context Resources" (below) and "SQLJ Connection Context Classes (-context)" on page 2-13 for related information.

Releasing Connection Context Resources

In some situations, you *must* use the `release()` method of an instance of a JPublisher-generated wrapper class in order to free SQLJ runtime connection context resources. This is true in the following set of circumstances:

- You used the SQLJ default setting `-codegen=iso` in translating SQLJ classes.

and:

- You use JDK 1.1.x and/or the SQLJ `runtime` library (as opposed to `runtime12`, `runtime11`, and so on) when you execute the generated class or classes.

and:

- You did not create the object with the constructor that takes an instance of `DefaultContext` (or some other connection context class you specified through the `-context` option when you ran JPublisher).

and:

- You have called one or more wrapper methods on the wrapper instance.

and:

- You did *not* use the `setConnectionContext()` method of the wrapper instance to explicitly set a connection context instance.

In these circumstances, a connection context instance would have been created implicitly on the object and must explicitly be freed through the `release()` method before the object goes out of scope.

(When there is an explicit connection context instance, such as through an explicit constructor or use of the `setConnectionContext()` method, using `release()` is not necessary.)

JPublisher Generation of Java Classes

When `-methods=false`, or when SQL object types do not define any methods, JPublisher does not generate wrapper methods for object types. Furthermore, when `-methods=false`, JPublisher does not generate code for PL/SQL packages at all, because they are not useful without wrapper methods. (Note that when `-methods=false`, JPublisher exclusively generates `.java` files.)

JPublisher generates the same Java code for reference, VARRAY, and nested table types regardless of whether `-methods` is `false` or `true`.

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

To initialize your object, you can:

- Assign an already initialized Java object to your Java object.

- Retrieve a copy of a SQL object into your Java object. To do this, you can:

    - Use the SQL object as an `OUT` argument or as the function call return of a JPublisher-generated wrapper method in some other class.

    - Retrieve the SQL object through `#sql` statements you write.

    - Retrieve the SQL object through JDBC calls you write.

- Construct the Java object and initialize its data.

Unlike the constructors generated in `.sqlj` source files, the constructors generated in `.java` source files do not take a connection argument. Instead, when your object is passed to or returned from a `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. On the contrary, this is not always possible. An object might 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 `setXXX()` 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 only construct a null reference. All non-null references come from the database.

Once you have initialized your object, you can accomplish the following:

- Pass the object to wrapper methods in other classes.

- Use the object as a host variable in `#sql` statements.

- Use the object as a host variable in JDBC calls.

- 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, you can call the `getXXX()` and `setXXX()` accessor methods.

  - For a class that represents a VARRAY or nested table, you can 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 `getXXX()` and `setXXX()` accessor methods of a class generated for an object type, the `getValue()` and `setValue()` methods read and write SQL objects.

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` to the Oracle `getORAData()` methods in the classes `ArrayDataResultSet`, `OracleCallableStatement`, and `OracleResultSet` in the `oracle.jdbc` package. The Oracle JDBC driver uses the `ORADataFactory` object to create objects of your JPublisher-generated class.

In addition, classes representing VARRAYs and nested tables have a few 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 wrapper type to another.

The `RationalP` example, described in "Example: Using Classes Generated for Packages" on page 3-65, includes a class that was generated for an object type that does not have wrapper methods.

User-Written Subclasses of JPublisher-Generated Classes

You might want to enhance the functionality of a custom Java class generated by JPublisher by adding methods and transient fields.

One way to accomplish this is to add methods directly to the JPublisher-generated class. 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, your changes (that is, the methods you have added) will be overwritten. Even if you direct JPublisher output to a separate file, you will still need to merge your changes into the file.

The preferred way to enhance the functionality of a generated class is to extend the class—that is, treat the JPublisher-generated class as a superclass, write a subclass to extend its functionality, then map the object type to the subclass.

This section discusses how to accomplish this.

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 `JAddress` provides.

Under this scenario, you can use JPublisher to generate a custom Java class `JAddress`, then write a subclass, `MyAddress`, which contains the added functionality. You then use JPublisher to map `ADDRESS` objects to the `MyAddress` class, not to the `JAddress` class. JPublisher will also produce a reference class for `MyAddress`, not `JAddress`.

To do this, JPublisher must alter the code it generates in the following ways:

- JPublisher will generate the reference class `MyAddressRef`, rather than `JAddressRef`.

- JPublisher will use 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`.

- JPublisher will use the `MyAddress` factory, instead of the `JAddress` factory, when the `ORADataFactory` interface is used to construct Java objects whose SQL type is `ADDRESS`.

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_class*:*map_class*

For the above scenario, this would be:

-sql=ADDRESS:JAddress:MyAddress

See "Declaration of Object Types and Packages to Translate (-sql)" on page 2-20 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 2-26 for information about the INPUT file.

In this syntax, JAddress indicates the name of the class that JPublisher will generate (as JAddress.java), and MyAddress specifies the name of the class that you have written. JPublisher will map the object type ADDRESS to the MyAddress class, not to the JAddress class. Therefore, if you retrieve an object that has an ADDRESS attribute, this attribute will be created as an instance of MyAddress in Java. Or if you retrieve an ADDRESS object directly, you will retrieve it into an instance of MyAddress.

For an example of how you would use JPublisher to generate the JAddress class, see "Example: Generating a SQLData Class" on page 3-27.

Writing the Class that Extends the Generated Class

The class that you create (for example, MyAddress.java) must have the following features:

■   The class must have 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 must implement the ORAData interface or the SQLData interface. The simplest way to do this is to inherit the necessary methods from the superclass.

■   If you are extending an ORAData class, you must also implement the ORADataFactory interface, either in the same class or in a different one. For

example, you could have a class `Employee` that implements `ORAData` and a class `EmployeeFactory` that implements `ORADataFactory`.

Following is a sample implementation of the `ORADataFactory` `create()` method:

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

Alternatively, the following code shows a more efficient implementation, where an initialized `UserClass` instance is created through the `UserClass(boolean)` constructor, which you must define. This constructor is provided in JPublisher-generated code, including the superclass that `UserClass` extends. Using this constructor ensures that a `UserClass` instance is not needlessly created if the data object is null, or needlessly re-initialized if the data object is non-null.

```
protected UserClass(boolean init) { super(boolean); }
public ORAData create(Datum d, int sqlType) throws SQLException
{
    return (d==null) ? null : create(new UserClass(false),d,sqlType);
}
```

"Example: Generating a SQLData Class" on page 3-27 illustrates the preceding features.

Changes in User-Written Subclasses of Oracle9*i* JPublisher-Generated Classes

If you have been providing user-written subclasses for JPublisher-generated classes under Oracle8*i* JPublisher, you should be aware that there are a number of relevant changes in how Oracle9*i* JPublisher generates code. You would have to make changes in any applications written against the Oracle8*i* functionality if you want to use it under Oracle9*i*.

> **Note:** If you use the `-compatible=8i` option setting, you will not see the changes discussed here and your application will continue to build and work as before.
>
> In general, however, it is generally advisable to make the transformation to Oracle9*i* JPublisher functionality, because this will help insulate your user code from implementation details of JPublisher-generated classes.

Following are the changes:

- Replace use of the declared `_ctx` connection context field with use of the provided `getConnectionContext()` method. The `_ctx` field is no longer supported under Oracle9*i*.

- Replace the explicit implementation of the `create()` method with a call to a superclass `create()` method.

  Assume that in the example below, `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 CustomDatum create(Datum d, int sqlType) throws SQLException
  {
    return create(new UserClass(),d,sqlType);
  }
  ```

  In addition, in `.sqlj` files, JPublisher now generates a protected constructor with a boolean argument that specifies whether the object needs to be initialized or not:

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

You can use this to optimize the UserClass code as described in "Writing the Class that Extends the Generated Class" on page 1-45.

- In addition to the getConnectionContext() method, Oracle9*i* JPublisher provides a getConnection() method that can be used to obtain the JDBC connection associated with the object.

The setFrom() and setValueFrom() Methods

Oracle9*i* JPublisher provides the following utility methods in generated .sqlj files:

- setFrom(anotherObject)

  This 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 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.

JPublisher Support for Inheritance

This section primarily discusses inheritance support for `ORAData` types, explaining the following related topics:

- how JPublisher implements support for inheritance—specifically, why you must construct subclass objects before selecting instances of non-final types from the database (or middle-tier database cache)

- why a reference class for a subtype does not extend the reference class for the base type, and how you can convert from one reference type to another reference type (typically a subclass or superclass)

This information is followed by 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 (
...
);
```

And consider the following JPublisher command line to create corresponding Java classes:

```
jpub -user=scott/tiger -sql=PERSON:Person,STUDENT:Student -usertypes=oracle
```

In this example, JPublisher generates a `Person` class and a `Student` class. The `Student` class extends the `Person` class, because `STUDENT` is a subtype of `PERSON`.

So initialization can complete properly, create instances of these classes—at least the leaf classes—before using these mapped types in your code. For example:

```
new Person();
new Student();
```

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.

In general, to handle this kind of situation, the `create()` method of a custom Java class (regardless of whether the class was created by JPublisher) must be able to create instances of any subclass that represents a subtype of the SQL object type of the `oracle.sql.Datum` argument. This ensures that the actual type of the created Java object will match the actual type of the SQL object.

You might think that the code for the `create()` method in the root class of a custom Java class hierarchy must mention all its subclasses. But if this were the case, you would have to modify the code for a base class when writing or generating a new subclass. At best, this process would be prone to errors, and would not even be possible if the programmer did not have access to the source code for the Java classes being extended.

Code generated by JPublisher solves this problem by creating a static initialization block in each subclass in 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 carries a small penalty—the static initialization blocks of the subclasses must be executed before the class hierarchy can be used to read objects from the database (or middle-tier database cache). 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 will invoke the constructor for its immediate superclass.

To better understand how code generated by JPublisher supports inheritance, try an example similar to the one at the beginning of this section, and look at the generated code.

ORAData Reference Types and Inheritance

This section explains why a custom reference class generated for a subtype by JPublisher does not extend the reference classes of the base type, and offers a workaround for how to convert from one reference type to another.

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 the example given in the previous section, repeated here for convenience:

```
CREATE TYPE PERSON AS OBJECT (
...
) NOT FINAL;

CREATE TYPE STUDENT UNDER PERSON (
...
);

jpub -user=scott/tiger -sql=PERSON:Person,STUDENT:Student -usertypes=oracle
```

In addition to generating `Person.sqlj` (or `.java`) and `Student.sqlj` (or `.java`), JPublisher will generate `PersonRef.java` and `StudentRef.java`.

Because the `Student` class extends the `Person` class, you might 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` can do something that a `StudentRef` cannot do: modify a `Person` object in the database.

The most important methods of the `PersonRef` class would be the following:

- `Person getValue()`
- `void setValue(Person c)`

The corresponding methods of the `StudentRef` class would be 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.

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. This section provides code to show you how to accomplish such a conversion in your program.

To convert from the reference type `XxxxRef` to the reference type `YyyyRef`, for example, use the following code:

```
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 comprises 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: Converting Between Reference Types" below provides sample code for such a conversion.

> **Note:** This conversion does not involve any type checking. Whether this conversion is actually permitted depends on your application and on the SQL schema you are using.

Example: 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
```

Java Class Here is a Java class with an example of reference type conversion as
discussed above, in "Converting Between Reference Types" on page 1-52.

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sqlj.runtime.Oracle;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ResultSetIterator;

public class Inheritance
{
  public static void main(String[] args) throws SQLException
  {
    System.out.println("Connecting.");
    Oracle.connect("jdbc:oracle:oci:@","scott","tiger");

    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;

        System.out.println("Selecting a person.");
        #sql { select ref(p) INTO :p_ref FROM PERSON_TAB p WHERE p.NAME='Larry' };

        System.out.println("Selecting an instructor.");
        #sql { select ref(p) INTO :i_ref FROM PERSON_TAB p WHERE p.NAME='Smith' };

        System.out.println("Selecting a part time instructor.");
        #sql { select ref(p) INTO :ipt_ref FROM PERSON_TAB p WHERE p.NAME='Myers' };

        System.out.println("Selecting a student.");
        #sql { select ref(p) INTO :s_ref FROM PERSON_TAB p WHERE p.NAME='John' };

        System.out.println("Selecting a part time student.");
        #sql { select ref(p) INTO :spt_ref FROM PERSON_TAB p WHERE p.NAME='Dave' };

        System.out.println("Selecting a graduate student.");
        #sql { select ref(p) INTO :g_ref FROM PERSON_TAB p WHERE p.NAME='Lisa' };

        // Connection object for conversions
        Connection conn = DefaultContext.getDefaultContext().getConnection();

        // 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);

        // 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);

        // Assigning a graduate ref to an part time instructor ref
        // ==> this should actually bomb at runtime!
        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);
        Oracle.close();
    }
}
```

SQLData Object Types and Inheritance

As described earlier, if you use the JPublisher `-usertypes=jdbc` setting instead of `-usertypes=oracle`, the custom Java class that JPublisher generates will implement the standard `SQLData` interface instead of the Oracle `ORAData` interface. The `SQLData` `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 will follow the same hierarchy as the SQL types.

`SQLData` implementations do not, however, offer the implicit mapping intelligence that JPublisher automatically generates into `ORAData` classes (as described in "ORAData Object Types and Inheritance" on page 1-49).

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 *Oracle9i JDBC Developer's Guide and Reference* for information.) In a SQLJ application, use a type map resource that is similar in nature to a properties file. (See the *Oracle9i SQLJ Developer's Guide and Reference* for information.)

In addition, be aware that there is no support for strongly typed object references in a SQLData implementation. All object references are simple `java.sql.Ref` instances.

Effect of Using SQL FINAL, NOT FINAL, INSTANTIABLE, NOT INSTANTIABLE

This section discusses the effect on JPublisher-generated wrapper classes of using the SQL modifiers `FINAL`, `NOT FINAL`, `INSTANTIABLE`, 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 the generated Java wrapper class through subclassing 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. Thus you must cast to some wrapper class that corresponds to an instantiable SQL subtype in order to call such a method.

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 only be created through subclasses that correspond to instantiable SQL types.

JPublisher Limitations

This section summarizes limitations in the current release of JPublisher.

- lack of support for some datatypes

    JPublisher does not currently support the following SQL and PL/SQL argument types:

    - BOOLEAN

    - NCHAR

    - NVARCHAR2

    - NCLOB

    - RECORD types

    - index-by tables

    - strongly-typed REF CURSOR

    - SQLJ object types

    - OPAQUE types

    JPublisher will not generate code for wrapper methods that use one or more of the unsupported datatypes. Instead, JPublisher will display one or more error messages.

    For more information about datatype support, see "Datatype Mapping Tables" on page 1-27.

- INPUT file error reporting

    JPublisher reports most, but not all, errors in the INPUT file. The few errors in the INPUT file that are not reported by JPublisher are described in "INPUT File Precautions" on page 2-32.

2

Command-Line Options and Input Files

This chapter describes the use of JPublisher option settings and input files to specify program behavior. The following topics are covered:

- JPublisher Options
- JPublisher Input Files

JPublisher Options

This section lists and discusses the use of JPublisher command-line options, covering the following topics:

- JPublisher Option Summary
- JPublisher Option Tips
- Notational Conventions
- Detailed Descriptions—Options That Affect Datatype Mappings
- Detailed Descriptions—General JPublisher Options

JPublisher Option Summary

Table 2–1 lists the options that you can use on the JPublisher command line, their syntax, and a brief description. The abbreviation "n/a" represents "not applicable".

Table 2–1   Summary of JPublisher Options

Option Name	Description	Default Value
-access	Determines the access modifiers that JPublisher includes in generated method definitions.	public
-builtintypes	Specifies the datatype mappings (jdbc or oracle) for non-numeric, non-LOB built-in datatypes.	jdbc
-case	Specifies the case of Java identifiers that JPublisher generates.	mixed
-compatible	Specifies the general Oracle8*i* compatibility mode, or the particular interface to implement in generated classes for Oracle mapping of user-defined types—ORAData or CustomDatum (supported for backwards compatibility); modifies the behavior of -usertypes=oracle.	oradata
-context	Specifies what JPublisher uses for connection contexts—the SQLJ DefaultContext class, a user-specified class, or a JPublisher-generated inner class.	DefaultContext

Table 2–1 Summary of JPublisher Options (Cont.)

Option Name	Description	Default Value
`-dir`	Specifies the directory that holds generated files/packages (an empty directory name results in all generated files being placed in the current directory; a non-empty directory name specifies a directory to be used as the root directory of a class hierarchy).	empty
`-driver`	Specifies the driver class that JPublisher uses for JDBC connections to the database.	`oracle.jdbc.OracleDriver`
`-encoding`	Specifies the Java encoding of JPublisher input files and output files.	the value of the System property `file.encoding`
`-input`	Specifies the file that lists the types and packages JPublisher translates.	n/a
`-lobtypes`	Specifies the datatype mappings (`jdbc` or `oracle`) that JPublisher uses for `BLOB` and `CLOB` types.	`oracle`
`-mapping`	Specifies which object attribute type and method argument type mapping the generated methods support. Note: This is deprecated in favor of the "XXXtypes" mapping options, but is supported for backwards compatibility.	`objectjdbc`
`-methods`	Determines whether JPublisher generates wrapper methods for SQL object methods and PL/SQL package methods. (As secondary effects, determines whether JPublisher generates `.sqlj` files or `.java` files, and whether it generates PL/SQL wrapper classes at all.)	`all`
`-numbertypes`	Specifies the datatype mappings (`jdbc`, `objectjdbc`, `bigdecimal`, or `oracle`) JPublisher uses for numeric datatypes.	`objectjdbc`
`-omit_schema_names`	Specifies whether all object types and package names JPublisher generates include the schema name.	do not omit schema names

Table 2–1   Summary of JPublisher Options (Cont.)

Option Name	Description	Default Value
-package	Specifies the name of the Java package into which JPublisher is generating Java wrappers.	n/a
-props	Specifies a file that contains JPublisher options in addition to those listed on the command line.	n/a
-sql	Specifies object types and packages for which JPublisher will generate code.	n/a
-types	Specifies object types for which JPublisher will generate code. Note: This option is deprecated in favor of -sql, but is supported for backwards compatibility.	n/a
-url	Specifies the URL JPublisher uses to connect to the database.	jdbc:oracle:oci:@ Note: With the Oracle9*i* release, use "oci" in the connect string for the Oracle JDBC OCI driver in any new code. For backwards compatibility, however, "oci8" and "oci7" are still accepted.
-user	Specifies an Oracle username and password.	n/a
-usertypes	Specifies the types mappings (jdbc or oracle) JPublisher uses for user-defined SQL types.	oracle

JPublisher Option Tips

Be aware of the following usage notes for JPublisher options:

- JPublisher always requires the -user option, either on the command line or in the properties file.

- Options are processed in the order in which they appear. Options from an INPUT file are processed at the point where the -input option occurs. Similarly, options from a properties file are processed at the point where the -props option occurs.

- If a particular option appears more than once, JPublisher uses the value from the last occurrence.

- It is advisable to specify a Java package for your generated classes, either on the command line, with the `-package` option, or in the properties file. For example, on the command line you could enter:

```
jpub -sql=Person -package=e.f ...
```

or in the properties file you could enter:

```
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 2-25 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's 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:

- Angle brackets `< . . . >` enclose strings that the user supplies.

- A vertical bar `|` separates alternatives within brackets.

- Terms in italics are like variables—specify an actual value or string.

- Square brackets `[ . . . ]` enclose optional items.

- Braces `{ . . . }` enclose a list of possible values—specify only one of the values within the braces.

- An ellipsis `. . .` immediately following an item (or items enclosed in brackets) means that you can repeat the item any number of times.

- Punctuation symbols other than those described above are entered as shown. These include `"."` and `"@"`, for example.

The next section discusses the options that affect datatype mappings. The remaining options are then discussed in alphabetical order.

Detailed Descriptions—Options That Affect 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).

- The -numbertypes option controls datatype mappings for numeric types.

- The -lobtypes option controls datatype mappings for the BLOB and CLOB 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. (Another, less flexible option, -mapping, is discussed later. 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 included with the object. The mappings control the types that the generated accessor methods support; that is, what types the getXXX() methods return and the setXXX() methods require.

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.

The -usertypes option 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. In addition, if -usertypes=oracle, you can use the -compatible option to specify using CustomDatum instead of ORAData for Oracle mapping. CustomDatum is replaced by ORAData and deprecated in Oracle9*i*, but is supported for backwards compatibility. (Beyond this, you can use the -compatible option to specify a more general Oracle8*i* compatibility mode. See "Oracle8i Compatibility Mode" on page 1-24.)

See "Details of Datatype Mapping" on page 1-26 for more information about the different datatype mappings and factors you should consider in deciding which mappings to use.

The following sections provide additional information about these type mapping options.

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. This is possibly used in conjunction with the -compatible option, described in "Backwards-Compatible Oracle Mapping for User-Defined Types (-compatible)" below.

When -usertypes=jdbc, JPublisher generates SQLData classes for object types. JPublisher does not generate classes for collection or object reference types in this case—use java.sql.Array for all collection types and java.sql.Ref for all object reference types.

> **Note:** The -usertypes=jdbc setting requires JDK 1.2 or higher, because the SQLData interface is a JDBC 2.0 feature.

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

-compatible={oradata|customdatum|8i}

If -usertypes=oracle, you have the option of setting -compatible=customdatum to implement the CustomDatum interface instead of the ORAData interface in your generated classes for user-defined types. CustomDatum is replaced by ORAData and deprecated in Oracle9*i*, but is still supported for backwards compatibility. If -usertypes=jdbc, a -compatible setting of customdatum (or oradata) is ignored.

Alternatively, this option has another mode of operation. With a setting of -compatible=8i, you can specify the general Oracle8*i* compatibility mode. This not only uses the CustomDatum interface, but also generates the same code that would be generated by Oracle8*i* JPublisher, and is equivalent to setting other JPublisher options for backwards compatibility to Oracle8*i*. 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 1-24.

> **Note:** If you use JPublisher in an environment that does not
> support the ORAData interface (such as Oracle8*i* JDBC 8.1.7 or prior
> releases), 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.)

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:

- The JDBC mapping maps most numeric datatypes to Java primitive types such
  as int and float, and maps DECIMAL and NUMBER to
  java.math.BigDecimal.

- The Object JDBC mapping (the default) maps most numeric datatypes to Java
  wrapper classes such as java.lang.Integer and java.lang.Float, and
  maps DECIMAL and NUMBER to java.math.BigDecimal.

- The BigDecimal mapping maps all numeric datatypes to
  java.math.BigDecimal.

- The Oracle mapping maps all numeric datatypes to oracle.sql.NUMBER.

Table 2–2 lists the SQL and PL/SQL types affected by the -numbertypes option,
and shows their Java type mappings for -numbertypes=jdbc and
-numbertypes=objectjdbc (the default).

Table 2–2   Mappings for Types Affected by the -numbertypes Option

SQL or PL/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	short	java.lang.Integer
REAL	float	java.lang.Float
DOUBLE PRECISION, FLOAT	double	java.lang.Double

Table 2–2   Mappings for Types Affected by the -numbertypes Option (Cont.)

SQL or PL/SQL Datatype	JDBC Mapping Type	Object JDBC Mapping Type
DEC, DECIMAL, NUMBER, NUMERIC	java.math.BigDecimal	java.math.BigDecimal

Mappings For LOB Types (-lobtypes)

-lobtypes={jdbc|oracle}

The -lobtypes option controls datatype mappings for the LOB types. Table 2–3 shows how these types are mapped for -lobtypes=oracle (the default) and for -lobtypes=jdbc.

Table 2–3   Mappings for Types Affected by the -lobtypes Option

SQL or PL/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.

- The java.sql.Clob and java.sql.Blob interfaces are new in JDK 1.2. If you use JDK 1.1, you should not select -lobtypes=jdbc.

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 2–4 lists the datatypes affected by the -builtintypes option and shows their Java type mappings for -builtintypes=oracle and -builtintypes=jdbc (the default).

Table 2–4   Mappings for Types Affected by the -builtintypes Option

SQL or PL/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[ ]
DATE	oracle.sql.DATE	java.sql.Timestamp

Mappings for All Types (-mapping)

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

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

> **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 backwards compatibility.

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 2–5.

Table 2–5   Relation of -mapping Settings to Settings of Other Mapping Options

	-builtintypes=	-numbertypes=	-lobtypes=	-usertypes=
-mapping=oracle	oracle	oracle	oracle	oracle
-mapping=jdbc	jdbc	jdbc	oracle	oracle
-mapping=objectjdbc	jdbc	objectjdbc	oracle	oracle
-mapping=bigdecimal	jdbc	bigdecimal	oracle	oracle

> **Note:** Because options are processed in the order in which they appear on the command line, if the -mapping option precedes one of the specific type mapping options (-builtintypes, -lobtypes, -numbertypes, or -usertypes) 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, the specific type mapping option is ignored.

Detailed Descriptions—General JPublisher Options

This section discusses the remaining JPublisher options, for settings other than datatype mappings. Options in this section are in alphabetical order.

Method Access (-access)

-access={public|protected|package}

The -access option determines the access modifiers that JPublisher includes in generated attribute setter and getter methods and member methods on object type wrapper classes, as well as for wrapper methods on PL/SQL packages.

JPublisher uses the possible option settings as follows:

- public (default)—Results in methods being generated with the public access modifier.

- protected—Results in methods being generated with the protected access modifier.

- package—Results in the access modifier being omitted, which means that generated methods are local to the package.

You might want to use a setting of -access=protected or -access=package if you need to control the usage of the generated JPublisher wrapper classes. Perhaps you are providing your own customized versions of the wrappers 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:** Constructors of JPublisher-generated classes, as well as
> wrappers 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 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
getXXX() and setXXX() method names, arguments of generated method names,
and Java wrapper names.

Table 2–6 describes the possible values for the -case option.

Table 2–6   Values for the -case Option

-case Option Value	Description
mixed (default)	The first letter of every word-unit of a class name or every word-unit after the first word-unit of a method name is in uppercase. All other characters are in lower case. An underscore (_) or dollar sign ($), or any character that is 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 that is 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 that is 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 that is illegal in Java and issues a warning message.

For class or attribute names that you enter with the -sql option, or class names in
the INPUT file, JPublisher retains the case of the letters in the name, overriding the
-case option.

JPublisher will retain as written the case of the Java class identifier for an object type specified on the command line or in the INPUT file. For example, if the command line includes the following:

```
-sql=Worker
```

then JPublisher generates:

```
public class Worker ... ;
```

Or, if the entry in the INPUT file is written as:

```
SQL wOrKeR
```

then JPublisher will follow the case for the identifier as it was entered in the INPUT file and generate:

```
public class wOrKeR ... ;
```

SQLJ Connection Context Classes (-context)

```
-context={generated|DefaultContext|user-specified}
```

The `-context` option controls the connection context class that JPublisher may use, and possibly declare, for `.sqlj` wrappers for user-defined object types and PL/SQL packages.

The setting `-context=DefaultContext` is the default and results in any JPublisher-generated `.sqlj` source files using the SQLJ default connection context class—`sqlj.runtime.ref.DefaultContext`—for all connection contexts.

Alternatively, you can specify any 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-specified class setting, instances of that class must be used for output from the `getConnectionContext()` method or input to the `setConnectionContext()` method. See "Considerations in Using Connection Contexts and Connection Instances" on page 1-38 for information about these methods.

The setting `-context=generated` results in the following inner class declaration in all `.sqlj` files generated by JPublisher:

```
#sql static context _Ctx;
```

This means that each PL/SQL package and each object type wrapper uses its own SQLJ connection context class. (Also see "Use of Connection Contexts and Instances in SQLJ Code Generated by JPublisher" on page 1-37.)

There are the following benefits in using the `DefaultContext` setting or user-specified-class setting:

- No additional context classes are generated.

- You have greater flexibility if you translate and compile your `.sqlj` files in separate steps (translating with the SQLJ `-compile=false` setting). Assuming you are not using JDK 1.2-specific types (such as `java.sql.BLOB`, `CLOB`, `Struct`, `Ref`, or `Array`), the resulting `.java` files can be compiled under either JDK 1.1.x or under JDK 1.2.x or higher. This is *not* the case with the setting `-context=generated`, because SQLJ connection contexts in JDK 1.1.x use `java.util.Dictionary` instances for object type maps, while SQLJ connection contexts in JDK 1.2 or higher use `java.util.Map` instances.

A benefit of using the `generated` setting is that it permits full control over the way the SQLJ translator performs online checking. Specifically, every object type and every PL/SQL package can be checked against its own 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.

Note that using the user-specified-class setting gives you the flexibility of the `generated` setting while still giving you the advantages of the `DefaultContext` setting.

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

See the *Oracle9i SQLJ Developer's Guide and Reference* for general information about SQLJ connection contexts.

Output Directory for Generated Files (-dir)

```
-dir=<directory name>
```

A non-empty `-dir` option setting specifies the root of the directory tree within which JPublisher will place Java and SQLJ source files. JPublisher will nest generated packages in this directory. A setting of "." (a period, or "dot") specifies the current directory as the root of the directory tree.

The empty setting, however, installs all generated file directly into the current directory—there is no hierarchy in this case. This is the default setting, but you can also specify it explicitly as follows:

```
-dir=
```

If you specify a non-empty setting, JPublisher combines the directory, the package name given with the `-package` option, and any package name included in a SQL statement in the `INPUT` file to determine the specific directory within which it will generate a `.java` or `.sqlj` file. The "Name for Generated Packages (-package)" section on page 2-18 discusses this in more detail.

For example, in the following command line (which is a single wrap-around line):

```
jpub -user=scott/tiger -input=demoin -mapping=oracle -case=lower -sql=employee
-package=corp -dir=demo
```

the `demo` directory will be the base directory for packages JPublisher generates for object types you specify in the `INPUT` file `demoin`.

You can specify `-dir` on the command line or in a properties file. The default value for the `-dir` option is empty.

JDBC Driver Class for Database Connection (-driver)

```
-driver=<driver_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.

Java Character Encoding (-encoding)

```
-encoding=<name_of_character_encoding>
```

The `-encoding` option specifies the Java character encoding of the `INPUT` file JPublisher reads and the `.sqlj` and `.java` files 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 are not required to specify this option unless you specify an `-encoding` option when you invoke SQLJ and your Java compiler, in which case you should use the same `-encoding` option for JPublisher.

You can use the -encoding option to specify any character encoding that is 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://www.javasoft.com/products/jdk/1.1/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.

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

-input=<*filename*>

The -input option specifies the name of a file from which JPublisher reads the names of object types and PL/SQL packages to translate, and other information it needs for their translation. JPublisher translates each object type and package in the list. You can think of the INPUT file as a makefile for type declarations—it lists the types that need Java class definitions.

In some cases, JPublisher might 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 2-30.

If you do not specify any packages or object types in an INPUT file or on the command line, then JPublisher translates all object types and 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 2-26.

Generate Classes for Packages and Wrapper Methods for Methods (-methods)

`-methods=(true|all|always|named|some|false|none)`

The value of the `-methods` option determines whether JPublisher generates wrapper methods for methods in object types and PL/SQL packages.

For `-methods=true` or, equivalently, `-methods=all` (the default), JPublisher generates wrapper methods for all the methods in the object types and PL/SQL packages it processes. As of Oracle9*i* release 9.0.1, this results in generation of a `.sqlj` source file whenever the underlying SQL object or package actually defines methods, but a `.java` source if not. (In previous releases, `.sqlj` source files were always generated for a `true` or `all` setting.)

The `-methods=always` setting also results in wrapper methods being generated; however, for backwards compatibility to earlier JPublisher versions, this setting always results in `.sqlj` files being generated for all SQL object types, regardless of whether the types define methods.

For `-methods=named` or, equivalently, `-methods=some`, JPublisher generates wrapper methods only for the methods explicitly named in the INPUT file.

For `-methods=false` or, equivalently, `-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.

The default is `-methods=all`.

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

Omit Schema Name from Generated Names (-omit_schema_names)

`-omit_schema_names`

The presence of the `-omit_schema_names` option determines whether certain object type names generated by JPublisher include the schema name. 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 object types and packages you use are declared identically in the two schemas.

`ORAData` and `SQLData` classes generated by JPublisher include a `static final String` that names the SQL object type matching the generated class. When the code generated by JPublisher executes, the object type name in the generated code is used to locate the object type in the database. If the object 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 object type name does include the schema name, the type is looked up in that schema.

If you *do not* specify -omit_schema_names on the command line, every object type or wrapper name generated by JPublisher is qualified with a schema name.

If you *do* specify -omit_schema_names on the command line, an object type or wrapper name generated by JPublisher is qualified with a schema name only if:

- you declare the object type or wrapper in a schema other than the one to which JPublisher is connected

or:

- you declare the object type or wrapper with a schema name on the command line or INPUT file

That is, an object type or wrapper from another schema requires a schema name to identify it, and the use of a schema name with the type or package on the command line or INPUT file overrides the -omit_schema_names option.

Name for Generated Packages (-package)

-package=<*package_translation_syntax*>

The -package option specifies the name of the package JPublisher generates. The name of the package appears in a package declaration in each .java or .sqlj file. The directory structure also reflects the package name. An explicit name in the INPUT file, after the -sql option, overrides the value given to the -package option.

Example 1 If the command line includes the following:

-dir=/a/b -package=c.d -case=mixed

and the INPUT file contains the following line (and assuming the SQL type PERSON has methods defined on it):

SQL PERSON AS Person

then in the following cases, JPublisher creates the file /a/b/c/d/Person.sqlj:

-sql=PERSON:Person
-sql=PERSON
SQL PERSON AS Person
SQL PERSON

The `Person.sqlj` file contains (among other things) the following package declaration:

```
package c.d;
```

Example 2 Now assume the following is again in the command line:

```
-dir=/a/b -package=c.d -case=mixed
```

but is followed by specification of an `INPUT` file containing the following:

```
-sql=PERSON:e.f.Person
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 creates the file `a/b/e/f/Person.sqlj`, which includes the following package declaration:

```
package e.f;
```

If you do not supply a package name for a class by any of the means described in this section, then JPublisher will not supply a name for the package containing the class. In addition, JPublisher will not generate a package declaration, and it will put the file containing the declaration of the class in the directory specified by the `-dir` option.

Occasionally, JPublisher might need to translate a type not explicitly listed in the `INPUT` file, because the type is used by another type that must be translated. In this case, the file declaring the required type is placed in the default package named on the command line, in a properties file, or in the `INPUT` file. JPublisher does not translate non-specified packages, because packages do not have dependencies on other packages.

Input Properties File (-props)

```
-props=<filename>
```

The `-props` option, entered on the command line, specifies the name of a JPublisher properties file that lists the values of commonly used options. JPublisher processes the properties file as if its contents were inserted in sequence on the command line at that point.

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 2-25.

> **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 -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.

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

-sql=<*object type and package translation syntax*>

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. Currently, JPublisher supports translation of object types and packages. JPublisher also translates the top-level subprograms in a schema, just as it does for subprograms in a PL/SQL package.

You can mix object types 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 will translate all the types and packages in the schema to which you are connected.

In this section, the -sql option is explained by translating it to the equivalent INPUT file syntax. INPUT file syntax is explained in "Understanding the Translation Statement" on page 2-26.

The JPublisher command-line syntax for -sql lets you indicate three possible type translations.

- -sql=*name_a*

  JPublisher interprets this syntax as: SQL *name_a*

- -sql=*name_a:name_c*

JPublisher interprets this syntax as: SQL *name_a* AS *name_c*

- -sql=*name_a*:*name_b*:*name_c*

JPublisher interprets this syntax as:
SQL *name_a* GENERATE *name_b* AS *name_c*

In this case, *name_a* must represent an object type.

> **Important:**   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 2-26 for information.

> **Note:**   The *name_a*:*name_b*:*name_c* translation syntax is not meaningful when *name_a* represents a package.

You enter -sql=... only once on the command line or properties file, followed by one or more object types and packages (including top-level "packages") that you want JPublisher to translate. If you enter more than one item for translation, they must be separated by commas, without any white space. This example assumes that CORPORATION is a package, and EMPLOYEE and ADDRESS are object types:

```
-sql=CORPORATION,EMPLOYEE:oracleEmployee,ADDRESS:JAddress:MyAddress
```

JPublisher will interpret this as follows:

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

And JPublisher executes the following:

- It creates a wrapper for the CORPORATION package.

- It translates the object type EMPLOYEE as oracleEmployee.

- It translates ADDRESS as JAddress, generating code so that ADDRESS objects will be represented by the MyAddress class that you will write to extend JAddress.

- It creates the references to the `MyAddress` class that you will write to extend `JAddress`.

If you want JPublisher to translate all the top-level PL/SQL subprograms in the schema to which JPublisher is connected, 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, known as `toplevel`, for the top level subprograms. If you want the class to be generated with a different name, you can declare the name with the `-sql=`*`name_a`*`:`*`name_b`* syntax. For example:

```
-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:

```
-sql=<schema-name>.toplevel
```

where `<`*`schema-name`*`>` is the name of the schema containing the top-level subprograms.

Declare 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 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 with the `-types` or `-sql` options, then JPublisher will translate 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_b AS name_c`

- `-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` syntax has the same functionality as `SQL`, `SQL...AS` and `SQL...GENERATE...AS` syntax. See "Understanding the Translation Statement" on page 2-26.

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, they must be separated by commas without any white space. For example, if you enter:

```
-types=CORPORATION,EMPLOYEE:oracleEmployee,ADDRESS:JAddress:MyAddress
```

JPublisher will interpret this as:

```
TYPE CORPORATION
TYPE EMPLOYEE AS oracleEmployee
TYPE ADDRESS GENERATE JAddress AS MyAddress
```

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:@
```

You can follow the "`@`" symbol with an Oracle SID.

To specify the Thin driver, enter:

```
-url=jdbc:oracle:thin:@host:port:sid
```

where `host` is the name of the host on which the database is running, `port` is the port number and `sid` is the Oracle SID.

> **Note:** With the Oracle9*i* release, use "oci" in the connect string for the Oracle JDBC OCI driver in any new code. For backwards compatibility, however, "oci8" and "oci7" are still accepted.

User Name and Password for Database Connection (-user)

```
-user=<username>/<password>
```

JPublisher requires the `-user` option, which specifies an Oracle user name and password. 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 username `scott` and password `tiger`:

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

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 where you can specify frequently-used options. You specify the name of the properties file on the JPublisher command line with the -props option. JPublisher processes the properties file as if its contents were inserted on the command line at that point.

In a properties file, you enter one (and only one) option with its associated value on each line. Enter the option name with the following prefix (including the period):

```
jpub.
```

You cannot use any white space within a line. You can enter any option *except* the -props option in the properties file.

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 was specified. If you specify an option more than once, JPublisher uses the last value encountered.

For example, consider the following command line (a single wrap-around line):

```
jpub -user=scott/tiger -sql=employee -mapping=oracle -case=lower -package=corp
-dir=demo
```

This is equivalent to the following:

```
jpub -props=my_properties
```

where my_properties is as follows:

```
jpub.user=scott/tiger
jpub.sql=employee
jpub.mapping=oracle
jpub.case=lower
jpub.package=corp
jpub.dir=demo
```

> **Note:** You must include the `jpub.` prefix (including the period) at the beginning of each option name. If you enter anything else before the option name, JPublisher will ignore the entire line.

"JPublisher Options" on page 2-2 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 the object types and PL/SQL packages 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 object types and packages, an INPUT file allows you a finer degree of control over how JPublisher translates object types and PL/SQL packages.

If you do not specify types or packages to translate in an INPUT file or on the command line, then JPublisher translates all object 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 object types and PL/SQL packages that you want JPublisher to translate. The translation statement can also optionally 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:

```
(SQL <name> | SQL [<schema_name>.]toplevel | 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>* Clause Enter SQL *<name>* to identify an object type or a PL/SQL package that you want JPublisher to translate. JPublisher examines the *<name>*, determines whether it is an object type or a package name, 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 in Oracle9*i*.

You can enter `<name>` as `<schema_name>.<name>` to specify the schema to which the object type or package belongs. If you enter `<schema_name>.toplevel`, JPublisher translates the top-level subprograms in schema `<schema_name>`.

> **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
> ```
>
> Or, if also specifying a non-case-sensitive schema name:
>
> ```
> SQL SCOTT."CaseSensitiveType" AS CaseSensitiveType
> ```
>
> Or, if also specifying a case-sensitive schema name:
>
> ```
> SQL "Scott"."CaseSensitiveType AS CaseSensitiveType
> ```
>
> (The `AS` clauses, described below, are optional.)
>
> Avoid situations where a dot (".") is part of the schema name or type name itself.

> **Note:** The `TYPE` syntax is currently supported for compatibility, but deprecated. Use the `SQL` syntax instead.

AS <*java_name_2*> Clause This clause optionally specifies the name of the Java class that represents the user-defined type or PL/SQL package. The `<java_name_2>` can be any legal Java name and can include a package identifier. The case of the Java name overrides the value of the `-case` option. For more information on how to name packages, see "Package Naming Rules in the INPUT File" on page 2-29.

When you use the `AS` clause without a `GENERATE` clause, the class in the `AS` clause is what JPublisher generates and is mapped to the SQL type.

When you use the `AS` clause *with* a `GENERATE` clause, JPublisher generates the class in the `GENERATE` clause but maps the SQL type to the class in the `AS` clause. You

manually create the class in the AS clause, extending the class that JPublisher generates.

Also see "Extending JPublisher-Generated Classes" on page 1-44.

GENERATE *<java_name_1>* Clause   This clause specifies the name of the class that JPublisher generates when you want to create a subclass for mapping purposes. Use the GENERATE clause in conjunction with the AS clause. JPublisher generates the class in the GENERATE clause. The AS clause specifies the name of the subclass that you create and that your Java program will use to represent the SQL object type.

The <*java_name_1*> can be any legal Java name and can include a package identifier. Its case overrides the value of the -case option.

Use the GENERATE clause only when you are translating object types. When you are translating an object type, the code JPublisher generates mentions both the name of the class that JPublisher generates and the name of the class that your Java program will use to represent the SQL object type. When these are two different classes, use GENERATE...AS.

Do not use this clause if you are translating PL/SQL packages. When you are translating a PL/SQL package, the code JPublisher generates mentions only the name of the class that JPublisher generates, so there is no need to use the GENERATE clause in this case.

Also see "Extending JPublisher-Generated Classes" on page 1-44.

TRANSLATE *<database_member_name>* AS *<simple_java_name>* Clause   This clause optionally specifies a different name for an attribute or method. The <*database_member_name*> is the name of an attribute of a type, or a method of a type or package, which is to be translated to the following <*simple_java_name*>. The <*simple_java_name*> can be any legal Java name, and its case overrides the value of 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 as modified according to the value of the -case option. Reasons why you might want to rename an attribute name 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 `getXXX()` and `setXXX()` method names, so you might want to capitalize the first letter of your attribute names. For example, if you enter:

```
TRANSLATE FIRSTNAME AS FirstName
```

JPublisher will generate a `getFirstName()` method and a `setFirstName()` method. In contrast, if you enter:

```
TRANSLATE FIRSTNAME AS firstName
```

JPublisher will generate 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   If you use a simple Java identifier to name a class in the `INPUT` file, its full class name will include the package name from the `-package` option. If the class name in the `INPUT` file is qualified with a package name, then that package name overrides the value of the `-package` option and becomes the full package name of the class.

For example:

- If you enter the syntax:

    ```
    SQL A AS B
    ```

    then JPublisher uses the value that was entered for `-package` on the command line or the properties file.

- If you enter the syntax:

```
SQL A AS B.C
```

then JPublisher interprets B.C to represent the full class name.

For example, if you enter:

```
-package=a.b
```

on the command line and the INPUT file contains the translation statement:

```
SQL scott.employee AS e.Employee
```

then JPublisher will generate the class as:

```
e.Employee
```

For more examples of how the package name is determined, see "Name for Generated Packages (-package)" on page 2-18.

Translating Additional Types It might 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 other types as necessary. Recall the example in "Sample JPublisher Translation" on page 1-15. Assume 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 packages you do not request. Because packages do not have attributes, they do not have any dependencies on other packages.

Sample Translation Statement

To better illustrate the function of the INPUT file, consider a more complicated version of the example in "Sample JPublisher Translation" on page 1-15. Consider the following command line (a single wrap-around line):

```
jpub -user=scott/tiger -input=demoin -dir=demo -numbertypes=oracle -package=corp
-case=same
```

The INPUT file demoin now contains:

```
SQL employee AS c.Employee
  TRANSLATE NAME AS Name
            HIRE_DATE AS HireDate
```

The -case=same option indicates that generated Java identifiers should maintain the same case as in the database. Any identifier in a CREATE TYPE or CREATE PACKAGE declaration is stored in upper case in the database unless it is quoted. However, the -case option is applied only to those identifiers not explicitly mentioned in the INPUT file. Thus, Employee will appear as written. The attribute identifiers not specifically mentioned (that is, EMPNO, DEPTNO, and SALARY) will remain in upper case, but JPublisher will translate the specifically mentioned NAME and HIRE_DATE attribute identifiers as shown.

The translation statement specifies a SQL object type to be translated. In this case, there is only one object type, Employee.

The AS c.Employee clause causes the package name to be further qualified. The translated type will be written to the following file:

```
./demo/corp/c/Employee.sqlj          (UNIX)
.\demo\corp\c\Employee.sqlj          (Windows NT)
```

(This assumes the object type defines methods; otherwise Employee.java will be generated instead.)

The generated file is written in package corp.c in output directory demo. Note that the package name is reflected in the directory structure.

The TRANSLATE...AS clause specifies that the name of any mentioned object attributes should be changed when the type is translated into a Java class. In this case, the NAME attribute is changed to Name and the HIRE_DATE attribute is changed to HireDate.

INPUT File Precautions

This section describes some of the common errors made in `INPUT` files. Check for these errors before you run JPublisher. Although JPublisher reports most of the errors that it finds in the `INPUT` file, it does not report these.

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 will silently overwrite the first. For example, if the `INPUT` file contains:

```
type PERSON1 as person
TYPE PERSON2 as person
```

JPublisher will create the file `person.java` for `PERSON1` and will then overwrite 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 will generate `getXXX()` and `setXXX()` 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 `getXXX()` methods with the same name and the two `setXXX()` 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 will ignore it without issuing a warning message.

For example, if the `INPUT` file contains: `type PERSON translate X as attr1`

and `X` is not an attribute of `PERSON`, JPublisher will not issue a warning message.

3

JPublisher Examples

This chapter provides examples of the output JPublisher produces when translating object types and PL/SQL packages. This chapter contains the following sections:

- Example: JPublisher Translations with Different Mappings contains examples of JPublisher output, comparing different outputs where only the values of the datatype mapping parameters are changed.

- Example: JPublisher Object Attribute Mapping illustrates an example of JPublisher output when translating different object types.

- Example: Generating a SQLData Class covers an example of JPublisher output when generating classes that implement the SQLData interface.

- Example: Extending JPublisher Classes presents an example of JPublisher output when generating a class that you will extend.

- Example: Wrappers Generated for Methods in Objects shows an example of JPublisher output when generating method wrappers for object type attributes and methods.

- Example: Wrappers Generated for Methods in Packages shows an example of JPublisher output when generating method wrappers for PL/SQL methods.

- Example: Using Classes Generated for Object Types presents a complete program using the classes that JPublisher generates for object types.

- Example: Using Classes Generated for Packages presents a complete program using the classes and method wrappers that JPublisher generates for objects and packages respectively.

- Example: Using Datatypes Not Supported by JDBC contains an example of how to write anonymous PL/SQL blocks that will let you employ datatypes not supported by JDBC.

Example: JPublisher Translations with Different Mappings

This section presents sample output from JPublisher with the only difference in the translations being the values of the datatype mapping parameters. It uses the SQL type declaration and JPublisher command line presented in "Sample JPublisher Translation" on page 1-15 (repeated here for convenience).

Type declaration:

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

Command line (a single wrap-around line):

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

In the following two examples, JPublisher translates the types using different datatype mapping options:

- first, with -numbertypes=jdbc and -builtintypes=jdbc
- second, with -numbertypes=oracle and -builtintypes=oracle.

JPublisher Translation with the JDBC Mapping

The SQL program presented in "Sample JPublisher Translation" on page 1-15 is translated here by JPublisher with -numbertypes=jdbc. No other changes have been made to the command line.

Because the user requests the JDBC mapping rather than the Object JDBC mapping for numeric types, the getXXX() and setXXX() accessor methods use the type int instead of Integer and the type float instead of Float.

Following are the contents of the Employee.java file. The EmployeeRef.java file is unchanged because it does not depend on the types of the attributes.

> **Note:** The details of method bodies generated by JPublisher might change in future releases.

```
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;

  static int[] _sqlType =
  {
    12, 4, 2, 91, 7
  };

  static ORADataFactory[] _factory = new ORADataFactory[5];

  static final Employee _EmployeeFactory = new Employee();
  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); }

  /* 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 int getEmpno() throws SQLException
    { return ((Integer) _struct.getAttribute(1)).intValue(); }

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

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

}
```

JPublisher Translation with the Oracle Mapping

The SQL program presented in "Sample JPublisher Translation" on page 1-15 is translated here by JPublisher with -numbertypes=oracle and -builtintypes=oracle. No other changes have been made to the command line.

Because the user requests Oracle type mappings, the getXXX() and setXXX() accessor methods employ the type oracle.sql.CHAR instead of String, the type oracle.sql.DATE instead of java.sql.Timestamp, and the type oracle.sql.NUMBER instead of Integer, java.math.BigDecimal, and Float.

Following are the contents of the Employee.java file. The EmployeeRef.java file is unchanged, because it does not depend on the types of the attributes.

> **Note:** The details of method bodies that JPublisher generates might change in future releases.

```
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;

static int[] _sqlType =
{
  12, 4, 2, 91, 7
};

static ORADataFactory[] _factory = new ORADataFactory[5];

static final Employee _EmployeeFactory = new Employee();
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); }

/* 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 oracle.sql.CHAR getName() throws SQLException
{ return (oracle.sql.CHAR) _struct.getOracleAttribute(0); }

public void setName(oracle.sql.CHAR name) throws SQLException
{ _struct.setOracleAttribute(0, name); }
```

```
public oracle.sql.NUMBER getEmpno() throws SQLException
{ return (oracle.sql.NUMBER) _struct.getOracleAttribute(1); }

public void setEmpno(oracle.sql.NUMBER empno) throws SQLException
{ _struct.setOracleAttribute(1, empno); }


public oracle.sql.NUMBER getDeptno() throws SQLException
{ return (oracle.sql.NUMBER) _struct.getOracleAttribute(2); }

public void setDeptno(oracle.sql.NUMBER deptno) throws SQLException
{ _struct.setOracleAttribute(2, deptno); }


public oracle.sql.DATE getHiredate() throws SQLException
{ return (oracle.sql.DATE) _struct.getOracleAttribute(3); }

public void setHiredate(oracle.sql.DATE hiredate) throws SQLException
{ _struct.setOracleAttribute(3, hiredate); }


public oracle.sql.NUMBER getSalary() throws SQLException
{ return (oracle.sql.NUMBER) _struct.getOracleAttribute(4); }

public void setSalary(oracle.sql.NUMBER salary) throws SQLException
{ _struct.setOracleAttribute(4, salary); }

}
```

Example: JPublisher Object Attribute Mapping

This section provides examples of JPublisher output for a variety of object attribute types, demonstrating the various datatype mappings that JPublisher creates.

The example defines an address object (address) and then uses it as the basis of the definition of an address array (Addr_Array). The alltypes object definition also uses the address and address array objects to demonstrate the mappings that JPublisher creates for object references and arrays (see attr17, attr18, and attr19 in the alltypes object definition below).

```
CONNECT SCOTT/TIGER;

CREATE OR REPLACE TYPE address AS object
(
  street varchar2(50),
  city   varchar2(50),
  state  varchar2(30),
  zip    number
);

CREATE OR REPLACE TYPE Addr_Array AS varray(10) OF address;
CREATE OR REPLACE TYPE ntbl AS table OF Integer;
CREATE TYPE alltypes AS object (
  attr1  bfile,
  attr2  blob,
  attr3  char(10),
  attr4  clob,
  attr5  date,
  attr6  decimal,
  attr7  double precision,
  attr8  float,
  attr9  integer,
  attr10 number,
  attr11 numeric,
  attr12 raw(20),
  attr13 real,
  attr14 smallint,
  attr15 varchar(10),
  attr16 varchar2(10),
  attr17 address,
  attr18 ref address,
  attr19 Addr_Array,
  attr20 ntbl);
```

In this example, JPublisher was invoked with the following command line (a single wrap-around line):

```
jpub -user=scott/tiger -input=demoin -dir=demo -package=corp -mapping=objectjdbc
-methods=false
```

> **Note:** The `-mapping` option, while deprecated, is still supported so is therefore demonstrated. The `-mapping=objectjdbc` setting is equivalent to the combination of `-builtintypes=jdbc`, `-numbertypes=objectjdbc`, `-lobtypes=oracle`, and `-usertypes=oracle`. See "Mappings for All Types (-mapping)" on page 2-10 for more information.

It is not necessary to create the `demo` and `corp` directories in advance. JPublisher will create the directories for you.

The `demoin` file contains these declarations:

```
SQL ADDRESS AS Address
SQL ALLTYPES AS all.Alltypes
```

JPublisher generates declarations of the types `Alltypes` and `Address`, because `demoin` explicitly lists them. It also generates declarations of the types `ntbl` and `AddrArray`, because the `Alltypes` type requires them.

Additionally, JPublisher generates declarations of the types `AlltypesRef` and `AddressRef`, because it generates a declaration of a reference type for each object type. A reference type is in the same package as the corresponding object type. Reference types are not listed in the `INPUT` file or on the command line. The `Address` and `AddressRef` types are in package `corp`, because `-package=corp` appears on the command line. The `Alltypes` and `AlltypesRef` types are in package `all`, because the `all` in `all.Alltypes` overrides `-package=corp`. The remaining types were not explicitly mentioned, so they go in package `corp`, which was specified on the command line.

Therefore, JPublisher creates the following files in package `corp`:

```
./demo/corp/Address.java
./demo/corp/AddressRef.java
./demo/corp/Ntbl.java
./demo/corp/AddrArray.java
```

and the following files in package `all`:

```
./demo/all/Alltypes.java
./demo/all/AlltypesRef.java
```

Listing and Description of Address.java Generated by JPublisher

The file `./demo/corp/Address.java` reads as follows:

> **Note:** The details of method bodies that JPublisher generates might change in future releases.

```java
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 Address implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  static int[] _sqlType =
  {
    12, 12, 12, 2
  };

  static ORADataFactory[] _factory = new ORADataFactory[4];

  static final Address _AddressFactory = new Address();
  public static ORADataFactory getORADataFactory()
  {
    return _AddressFactory;
  }
```

```
/* constructor */
protected Address(boolean init)
{ if(init) _struct = new MutableStruct(new Object[4], _sqlType, _factory); }
public Address()
{ this(true); }

/* 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(Address o, Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
   if (o == null) o = new Address(false);
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  return o;
}

/* accessor methods */
public String getStreet() throws SQLException
{ return (String) _struct.getAttribute(0); }

public void setStreet(String street) throws SQLException
{ _struct.setAttribute(0, street); }


public String getCity() throws SQLException
{ return (String) _struct.getAttribute(1); }

public void setCity(String city) throws SQLException
{ _struct.setAttribute(1, city); }


public String getState() throws SQLException
{ return (String) _struct.getAttribute(2); }

public void setState(String state) throws SQLException
{ _struct.setAttribute(2, state); }
```

```
public java.math.BigDecimal getZip() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(3); }

public void setZip(java.math.BigDecimal zip) throws SQLException
{ _struct.setAttribute(3, zip); }

}
```

The `Address.java` file illustrates several points about Java source files.
JPublisher-generated files begin with a package declaration whenever the generated
class is in a named package. Note that you can specify a package in any of these
ways:

- a `-package` parameter that you specify on the command line or in the
  properties file

- the `AS` *`<Java_identifier>`* clause in the `INPUT` file, where
  `Java_identifier` includes a package name

Import declarations for specific classes and interfaces mentioned by the `Address`
class follow the `package` declaration.

The class definition follows the `import` declarations. All classes JPublisher
generates are declared `public`.

SQLJ uses the `_SQL_NAME` and `_SQL_TYPECODE` strings to identify the SQL type
matching the `Address` class.

The no-argument constructor is used to create the `_AddressFactory` object,
which will be returned by `getORADataFactory()`. For efficiency, JPublisher also
generates a protected boolean constructor for `Address` objects. This can be used in
subclasses of `Address` to create uninitialized `Address` objects. Other `Address`
objects are constructed by the `create()` method. The protected `create(…,…,…)`
method is used to encapsulate details of the JPublisher implementation in the
JPublisher-generated `Address` class, and to simplify the writing of user-provided
subclasses. Implementation details, such as generation of the static `_factory` field
and the `_struct` field, are implementation-specific and should not be referenced or
exploited by any subclass of `Address`. (In this implementation, the `_factory` field
is an array of factories for attributes of `Address`, but in this case the factories are
null because none of the attribute types of `Address` require a factory. The `_struct`
field holds the object data and is a `MutableStruct` instance.)

The `toDatum()` method converts an `Address` object to a `Datum` object (in this case, a `STRUCT` object). JDBC requires the connection argument, although it might not be logically necessary.

The `getXXX()` and `setXXX()` accessor methods use the `objectjdbc` mapping for numeric attributes and the `jdbc` mapping for other attributes. The method names are in mixed case because `-case=mixed` is the default.

Listing of AddressRef.java Generated by JPublisher

The file `./demo/corp/AddressRef.java` reads as follows:

> **Note:** The details of method bodies that JPublisher generates might change in future releases.

```
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 AddressRef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;

  static final AddressRef _AddressRefFactory = new AddressRef();
  public static ORADataFactory getORADataFactory()
  {
    return _AddressRefFactory;
  }

  /* constructor */
  public AddressRef()
  {
  }
```

```
/* 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;
  AddressRef r = new AddressRef();
  r._ref = (REF) d;
  return r;
}

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

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

The `getValue()` method in the `AddressRef` class returns the address referenced
by an `AddressRef` object, with its proper type. The `setValue()` method copies
the contents of the `Address` argument into the database `Address` object to which
the `AddressRef` object refers.

Listing of Alltypes.java Generated by JPublisher

The file `./demo/all/Alltypes.java` reads as follows:

> **Note:** The details of method bodies that JPublisher generates
> might change in future releases.

```
package all;

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 Alltypes implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ALLTYPES";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  static int[] _sqlType =
  {
    -13, 2004, 1, 2005, 91, 3, 8, 6, 4, 2,
    3, -2, 7, 5, 12, 12, 2002, 2006, 2003, 2003
  };

  static ORADataFactory[] _factory = new ORADataFactory[20];
  static
  {
    _factory[16] = corp.Address.getORADataFactory();
    _factory[17] = corp.AddressRef.getORADataFactory();
    _factory[18] = corp.AddrArray.getORADataFactory();
    _factory[19] = corp.Ntbl.getORADataFactory();
  }

  static final Alltypes _AlltypesFactory = new Alltypes();
  public static ORADataFactory getORADataFactory()
  {
    return _AlltypesFactory;
  }

  /* constructor */
  protected Alltypes(boolean init)
  { if(init) _struct = new MutableStruct(new Object[20], _sqlType, _factory); }
  public Alltypes()
  { this(true); }

  /* 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(Alltypes o, Datum d, int sqlType) throws SQLException
                  {
                    if (d == null) return null;
                    if (o == null) o = new Alltypes(false);
                    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
                    return o;
                  }

                  /* accessor methods */
                  public oracle.sql.BFILE getAttr1() throws SQLException
                  { return (oracle.sql.BFILE) _struct.getOracleAttribute(0); }

                  public void setAttr1(oracle.sql.BFILE attr1) throws SQLException
                  { _struct.setOracleAttribute(0, attr1); }


                  public oracle.sql.BLOB getAttr2() throws SQLException
                  { return (oracle.sql.BLOB) _struct.getOracleAttribute(1); }

                  public void setAttr2(oracle.sql.BLOB attr2) throws SQLException
                  { _struct.setOracleAttribute(1, attr2); }


                  public String getAttr3() throws SQLException
                  { return (String) _struct.getAttribute(2); }

                  public void setAttr3(String attr3) throws SQLException
                  { _struct.setAttribute(2, attr3); }


                  public oracle.sql.CLOB getAttr4() throws SQLException
                  { return (oracle.sql.CLOB) _struct.getOracleAttribute(3); }

                  public void setAttr4(oracle.sql.CLOB attr4) throws SQLException
                  { _struct.setOracleAttribute(3, attr4); }


                  public java.sql.Timestamp getAttr5() throws SQLException
                  { return (java.sql.Timestamp) _struct.getAttribute(4); }
```

```
public void setAttr5(java.sql.Timestamp attr5) throws SQLException
{ _struct.setAttribute(4, attr5); }


public java.math.BigDecimal getAttr6() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(5); }

public void setAttr6(java.math.BigDecimal attr6) throws SQLException
{ _struct.setAttribute(5, attr6); }


public Double getAttr7() throws SQLException
{ return (Double) _struct.getAttribute(6); }

public void setAttr7(Double attr7) throws SQLException
{ _struct.setAttribute(6, attr7); }


public Double getAttr8() throws SQLException
{ return (Double) _struct.getAttribute(7); }

public void setAttr8(Double attr8) throws SQLException
{ _struct.setAttribute(7, attr8); }


public Integer getAttr9() throws SQLException
{ return (Integer) _struct.getAttribute(8); }

public void setAttr9(Integer attr9) throws SQLException
{ _struct.setAttribute(8, attr9); }


public java.math.BigDecimal getAttr10() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(9); }

public void setAttr10(java.math.BigDecimal attr10) throws SQLException
{ _struct.setAttribute(9, attr10); }


public java.math.BigDecimal getAttr11() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(10); }

public void setAttr11(java.math.BigDecimal attr11) throws SQLException
{ _struct.setAttribute(10, attr11); }
```

```
public byte[] getAttr12() throws SQLException
{ return (byte[]) _struct.getAttribute(11); }

public void setAttr12(byte[] attr12) throws SQLException
{ _struct.setAttribute(11, attr12); }


public Float getAttr13() throws SQLException
{ return (Float) _struct.getAttribute(12); }

public void setAttr13(Float attr13) throws SQLException
{ _struct.setAttribute(12, attr13); }


public Integer getAttr14() throws SQLException
{ return (Integer) _struct.getAttribute(13); }

public void setAttr14(Integer attr14) throws SQLException
{ _struct.setAttribute(13, attr14); }


public String getAttr15() throws SQLException
{ return (String) _struct.getAttribute(14); }

public void setAttr15(String attr15) throws SQLException
{ _struct.setAttribute(14, attr15); }


public String getAttr16() throws SQLException
{ return (String) _struct.getAttribute(15); }

public void setAttr16(String attr16) throws SQLException
{ _struct.setAttribute(15, attr16); }


public corp.Address getAttr17() throws SQLException
{ return (corp.Address) _struct.getAttribute(16); }

public void setAttr17(corp.Address attr17) throws SQLException
{ _struct.setAttribute(16, attr17); }


public corp.AddressRef getAttr18() throws SQLException
```

```
  { return (corp.AddressRef) _struct.getAttribute(17); }

  public void setAttr18(corp.AddressRef attr18) throws SQLException
  { _struct.setAttribute(17, attr18); }


  public corp.AddrArray getAttr19() throws SQLException
  { return (corp.AddrArray) _struct.getAttribute(18); }

  public void setAttr19(corp.AddrArray attr19) throws SQLException
  { _struct.setAttribute(18, attr19); }


  public corp.Ntbl getAttr20() throws SQLException
  { return (corp.Ntbl) _struct.getAttribute(19); }

  public void setAttr20(corp.Ntbl attr20) throws SQLException
  { _struct.setAttribute(19, attr20); }

}
```

When a declared class requires user-defined classes from another package, JPublisher generates import declarations for those user-defined classes following the import declaration for the oracle.sql package. In this case, JDBC requires the Address and AddressRef classes from package corp.

The attributes with types Address, AddressRef, AddrArray, and Ntbl require the construction of factories. The static block puts the correct factories in the _factory array.

> **Note:** Notice that the SMALLINT SQL type for attr14 maps to the Java type short, but this maps to Integer in -numbertypes=objectjdbc mapping. This was a JPublisher implementation decision. See "Mappings For Numeric Types (-numbertypes)" on page 2-8 for related information.

Listing of AlltypesRef.java Generated by JPublisher

The file `./demo/corp/all/AlltypesRef.java` reads as follows:

> **Note:** The details of method bodies that JPublisher generates might change in future releases.

```
package all;

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 AlltypesRef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.ALLTYPES";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;

  static final AlltypesRef _AlltypesRefFactory = new AlltypesRef();
  public static ORADataFactory getORADataFactory()
  {
    return _AlltypesRefFactory;
  }

  /* constructor */
  public AlltypesRef()
  {
  }

  /* 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;
  AlltypesRef r = new AlltypesRef();
  r._ref = (REF) d;
  return r;
}

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

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

Listing of Ntbl.java Generated by JPublisher

The file ./demo/corp/Ntbl.java reads as follows:

> **Note:** The details of method bodies that JPublisher generates
> might change in future releases.

```
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.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;

public class Ntbl implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.NTBL";
  public static final int _SQL_TYPECODE = OracleTypes.ARRAY;
```

```
MutableArray _array;

static final Ntbl _NtblFactory = new Ntbl();
public static ORADataFactory getORADataFactory()
{
  return _NtblFactory;
}

/* constructors */
public Ntbl()
{
  this((Integer[])null);
}

public Ntbl(Integer[] a)
{
  _array = new MutableArray(4, a, null);
}

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

/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Ntbl a = new Ntbl();
  a._array = new MutableArray(4, (ARRAY) d, null);
  return a;
}

public int length() throws SQLException
{
  return _array.length();
}

public int getBaseType() throws SQLException
{
  return _array.getBaseType();
}
```

```
public String getBaseTypeName() throws SQLException
{
  return _array.getBaseTypeName();
}

public ArrayDescriptor getDescriptor() throws SQLException
{
  return _array.getDescriptor();
}

/* array accessor methods */
public Integer[] getArray() throws SQLException
{
  return (Integer[]) _array.getObjectArray();
}

public void setArray(Integer[] a) throws SQLException
{
  _array.setObjectArray(a);
}

public Integer[] getArray(long index, int count) throws SQLException
{
  return (Integer[]) _array.getObjectArray(index, count);
}

public void setArray(Integer[] a, long index) throws SQLException
{
  _array.setObjectArray(a, index);
}

public Integer getElement(long index) throws SQLException
{
  return (Integer) _array.getObjectElement(index);
}

public void setElement(Integer a, long index) throws SQLException
{
  _array.setObjectElement(a, index);
}

}
```

Listing of AddrArray.java Generated by JPublisher

JPublisher generates declarations of the type `AddrArray` because they are required by the `Alltypes` type. The file `./demo/corp/AddrArray.java` reads as follows:

Note: The details of method bodies that JPublisher generates might change in future releases.

```
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.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;

public class AddrArray implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDR_ARRAY";
  public static final int _SQL_TYPECODE = OracleTypes.ARRAY;

  MutableArray _array;

  static final AddrArray _AddrArrayFactory = new AddrArray();
  public static ORADataFactory getORADataFactory()
  {
    return _AddrArrayFactory;
  }

  /* constructors */
  public AddrArray()
  {
    this((Address[])null);
  }

  public AddrArray(Address[] a)
  {
    _array = new MutableArray(2002, a, Address.getORADataFactory());
```

```
    }

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

    /* ORADataFactory interface */
    public ORAData create(Datum d, int sqlType) throws SQLException
    {
      if (d == null) return null;
      AddrArray a = new AddrArray();
      a._array = new MutableArray(2002, (ARRAY) d, Address.getORADataFactory());
      return a;
    }

    public int length() throws SQLException
    {
      return _array.length();
    }

    public int getBaseType() throws SQLException
    {
      return _array.getBaseType();
    }

    public String getBaseTypeName() throws SQLException
    {
      return _array.getBaseTypeName();
    }

    public ArrayDescriptor getDescriptor() throws SQLException
    {
      return _array.getDescriptor();
    }

    /* array accessor methods */
    public Address[] getArray() throws SQLException
    {
      return (Address[]) _array.getObjectArray(
        new Address[_array.length()]);
    }

    public void setArray(Address[] a) throws SQLException
```

```
    {
      _array.setObjectArray(a);
    }

    public Address[] getArray(long index, int count) throws SQLException
    {
      return (Address[]) _array.getObjectArray(index,
        new Address[_array.sliceLength(index, count)]);
    }

    public void setArray(Address[] a, long index) throws SQLException
    {
      _array.setObjectArray(a, index);
    }

    public Address getElement(long index) throws SQLException
    {
      return (Address) _array.getObjectElement(index);
    }

    public void setElement(Address a, long index) throws SQLException
    {
      _array.setObjectElement(a, index);
    }

}
```

Example: Generating a SQLData Class

This example is identical to the previous one, except that JPublisher generates a SQLData class rather than an ORAData class. The command line for this example is:

```
jpub -user=scott/tiger -input=demoin -dir=demo -package=corp -mapping=objectjdbc
-usertypes=jdbc -methods=false
```

(This is a single wrap-around command line.)

> **Note:** The -mapping option, while deprecated, is still supported so is therefore demonstrated. The -mapping=objectjdbc setting is equivalent to the combination of -builtintypes=jdbc, -numbertypes=objectjdbc, -lobtypes=oracle, and -usertypes=oracle; however, this command line overrides the -usertypes=oracle setting with a -usertypes=jdbc setting. See "Mappings for All Types (-mapping)" on page 2-10 for more information about the -mapping option.

The option -usertypes=jdbc instructs JPublisher to generate classes that implement the SQLData interface. The SQLData interface supports reference and collection classes generically, using the generic types java.sql.Ref and java.sql.Array rather than using custom classes. Therefore, JPublisher generates only two classes:

```
./demo/corp/Address.java
./demo/all/Alltypes.java
```

Listing of Address.java Generated by JPublisher

Because we specified -usertypes=jdbc in this example, the Address class implements the java.sql.SQLData interface rather than the oracle.sql.ORAData interface. The file ./demo/corp/Address.java reads as follows:

> **Note:** The details of method bodies that JPublisher generates might change in future releases.

```
package corp;
```

```
import java.sql.SQLException;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleTypes;
import java.sql.SQLData;
import java.sql.SQLInput;
import java.sql.SQLOutput;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Address implements SQLData
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  private String m_street;
  private String m_city;
  private String m_state;
  private java.math.BigDecimal m_zip;

  /* constructor */
  public Address()
  {
  }

  public void readSQL(SQLInput stream, String type)
  throws SQLException
  {
      setStreet(stream.readString());
      setCity(stream.readString());
      setState(stream.readString());
      setZip(stream.readBigDecimal());
  }

  public void writeSQL(SQLOutput stream)
  throws SQLException
  {
      stream.writeString(getStreet());
      stream.writeString(getCity());
      stream.writeString(getState());
      stream.writeBigDecimal(getZip());
  }

  public String getSQLTypeName() throws SQLException
  {
    return _SQL_NAME;
```

```
        }

        /* accessor methods */
        public String getStreet()
        { return m_street; }

        public void setStreet(String street)
        { m_street = street; }


        public String getCity()
        { return m_city; }

        public void setCity(String city)
        { m_city = city; }


        public String getState()
        { return m_state; }

        public void setState(String state)
        { m_state = state; }


        public java.math.BigDecimal getZip()
        { return m_zip; }

        public void setZip(java.math.BigDecimal zip)
        { m_zip = zip; }

    }
```

Listing of Alltypes.java Generated by JPublisher

Because -usertypes=jdbc was specified in this example, the Alltypes class
implements the java.sql.SQLData interface rather than the
oracle.sql.ORAData interface. Although the SQLData interface is a
vendor-neutral standard, there is Oracle-specific code in the Alltypes class
because it uses Oracle-specific types such as oracle.sql.BFILE and
oracle.sql.CLOB.

The file `./demo/corp/Alltypes.java` reads as follows:

> **Note:**   The details of method bodies that JPublisher generates
> might change in future releases.

```
package all;

import java.sql.SQLException;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleTypes;
import java.sql.SQLData;
import java.sql.SQLInput;
import java.sql.SQLOutput;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Alltypes implements SQLData
{
  public static final String _SQL_NAME = "SCOTT.ALLTYPES";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  private oracle.sql.BFILE m_attr1;
  private oracle.sql.BLOB m_attr2;
  private String m_attr3;
  private oracle.sql.CLOB m_attr4;
  private java.sql.Timestamp m_attr5;
  private java.math.BigDecimal m_attr6;
  private Double m_attr7;
  private Double m_attr8;
  private Integer m_attr9;
  private java.math.BigDecimal m_attr10;
  private java.math.BigDecimal m_attr11;
  private byte[] m_attr12;
  private Float m_attr13;
  private Integer m_attr14;
  private String m_attr15;
  private String m_attr16;
  private corp.Address m_attr17;
  private java.sql.Ref m_attr18;
  private java.sql.Array m_attr19;
  private java.sql.Array m_attr20;

  /* constructor */
```

```
   public Alltypes()
   {
   }

   public void readSQL(SQLInput stream, String type)
   throws SQLException
   {
       setAttr1((oracle.sql.BFILE)
((oracle.sql.OracleJdbc2SQLInput)stream).readOracleObject());
       setAttr2((oracle.sql.BLOB)
((oracle.sql.OracleJdbc2SQLInput)stream).readOracleObject());
       setAttr3(stream.readString());
       setAttr4((oracle.sql.CLOB)
((oracle.sql.OracleJdbc2SQLInput)stream).readOracleObject());
       setAttr5(stream.readTimestamp());
       setAttr6(stream.readBigDecimal());
       setAttr7(new Double(stream.readDouble()));
       if (stream.wasNull()) setAttr7(null);
       setAttr8(new Double(stream.readDouble()));
       if (stream.wasNull()) setAttr8(null);
       setAttr9(new Integer(stream.readInt()));
       if (stream.wasNull()) setAttr9(null);
       setAttr10(stream.readBigDecimal());
       setAttr11(stream.readBigDecimal());
       setAttr12(stream.readBytes());
       setAttr13(new Float(stream.readFloat()));
       if (stream.wasNull()) setAttr13(null);
       setAttr14(new Integer(stream.readInt()));
       if (stream.wasNull()) setAttr14(null);
       setAttr15(stream.readString());
       setAttr16(stream.readString());
       setAttr17((corp.Address) stream.readObject());
       setAttr18(stream.readRef());
       setAttr19(stream.readArray());
       setAttr20(stream.readArray());
   }

   public void writeSQL(SQLOutput stream)
   throws SQLException
   {
       ((oracle.sql.OracleSQLOutput)stream).writeOracleObject(getAttr1());
       ((oracle.sql.OracleSQLOutput)stream).writeOracleObject(getAttr2());
       stream.writeString(getAttr3());
       ((oracle.sql.OracleSQLOutput)stream).writeOracleObject(getAttr4());
       stream.writeTimestamp(getAttr5());
```

```
        stream.writeBigDecimal(getAttr6());
        if (getAttr7() == null)
          stream.writeBigDecimal(null);
        else
          stream.writeDouble(getAttr7().doubleValue());
        if (getAttr8() == null)
          stream.writeBigDecimal(null);
        else
          stream.writeDouble(getAttr8().doubleValue());
        if (getAttr9() == null)
          stream.writeBigDecimal(null);
        else
          stream.writeInt(getAttr9().intValue());
        stream.writeBigDecimal(getAttr10());
        stream.writeBigDecimal(getAttr11());
        stream.writeBytes(getAttr12());
        if (getAttr13() == null)
          stream.writeBigDecimal(null);
        else
          stream.writeFloat(getAttr13().floatValue());
        if (getAttr14() == null)
          stream.writeBigDecimal(null);
        else
          stream.writeInt(getAttr14().intValue());
        stream.writeString(getAttr15());
        stream.writeString(getAttr16());
        stream.writeObject(getAttr17());
        stream.writeRef(getAttr18());
        stream.writeArray(getAttr19());
        stream.writeArray(getAttr20());
      }

      public String getSQLTypeName() throws SQLException
      {
        return _SQL_NAME;
      }

      /* accessor methods */
      public oracle.sql.BFILE getAttr1()
      { return m_attr1; }

      public void setAttr1(oracle.sql.BFILE attr1)
      { m_attr1 = attr1; }
```

```
public oracle.sql.BLOB getAttr2()
{ return m_attr2; }

public void setAttr2(oracle.sql.BLOB attr2)
{ m_attr2 = attr2; }


public String getAttr3()
{ return m_attr3; }

public void setAttr3(String attr3)
{ m_attr3 = attr3; }


public oracle.sql.CLOB getAttr4()
{ return m_attr4; }

public void setAttr4(oracle.sql.CLOB attr4)
{ m_attr4 = attr4; }


public java.sql.Timestamp getAttr5()
{ return m_attr5; }

public void setAttr5(java.sql.Timestamp attr5)
{ m_attr5 = attr5; }


public java.math.BigDecimal getAttr6()
{ return m_attr6; }

public void setAttr6(java.math.BigDecimal attr6)
{ m_attr6 = attr6; }


public Double getAttr7()
{ return m_attr7; }

public void setAttr7(Double attr7)
{ m_attr7 = attr7; }


public Double getAttr8()
{ return m_attr8; }
```

```
public void setAttr8(Double attr8)
{ m_attr8 = attr8; }


public Integer getAttr9()
{ return m_attr9; }

public void setAttr9(Integer attr9)
{ m_attr9 = attr9; }


public java.math.BigDecimal getAttr10()
{ return m_attr10; }

public void setAttr10(java.math.BigDecimal attr10)
{ m_attr10 = attr10; }


public java.math.BigDecimal getAttr11()
{ return m_attr11; }

public void setAttr11(java.math.BigDecimal attr11)
{ m_attr11 = attr11; }


public byte[] getAttr12()
{ return m_attr12; }

public void setAttr12(byte[] attr12)
{ m_attr12 = attr12; }


public Float getAttr13()
{ return m_attr13; }

public void setAttr13(Float attr13)
{ m_attr13 = attr13; }


public Integer getAttr14()
{ return m_attr14; }

public void setAttr14(Integer attr14)
{ m_attr14 = attr14; }
```

```
public String getAttr15()
{ return m_attr15; }

public void setAttr15(String attr15)
{ m_attr15 = attr15; }


public String getAttr16()
{ return m_attr16; }

public void setAttr16(String attr16)
{ m_attr16 = attr16; }


public corp.Address getAttr17()
{ return m_attr17; }

public void setAttr17(corp.Address attr17)
{ m_attr17 = attr17; }


public java.sql.Ref getAttr18()
{ return m_attr18; }

public void setAttr18(java.sql.Ref attr18)
{ m_attr18 = attr18; }


public java.sql.Array getAttr19()
{ return m_attr19; }

public void setAttr19(java.sql.Array attr19)
{ m_attr19 = attr19; }


public java.sql.Array getAttr20()
{ return m_attr20; }

public void setAttr20(java.sql.Array attr20)
{ m_attr20 = attr20; }

}
```

Example: Extending JPublisher Classes

Here is an example of the scenario described in "Extending JPublisher-Generated Classes" on page 1-44.

The following is the code that you have written for the class `MyAddress.java` and stored in the directory `demo/corp`.

> **Note:** There is a way to code the `ORADataFactory` `create()` method to be more efficient than shown here, to ensure that an object instance is not needlessly created (if the data object is null) or needlessly re-initialized (if the data object is non-null). This is discussed in "Writing the Class that Extends the Generated Class" on page 1-45.

```
package corp;

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

public class MyAddress extends JAddress
{
  /* _SQL_NAME inherited from JAddress */
  /* _SQL_TYPECODE inherited from JAddress */

  static final MyAddress _MyAddressFactory = new MyAddress();
  public static ORADataFactory getORADataFactory()
  {
    return _MyAddressFactory;
  }

  /* constructor */
  public MyAddress()
  {
    super();
  }

  /* ORAData interface */
  /* toDatum() inherited from JAddress */
```

```
  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    return create(new MyAddress(), d, sqlType);
  }

  /* accessor methods inherited from JAddress */

  /* Additional methods go here. These additional methods (not shown)
     are the reason that JAddress was extended.
  */
}
```

To have JPublisher generate code for the JAddress class, recognizing that MyAddress extends JAddress, enter this command line:

```
jpub –user=scott/tiger –input=demoin –dir=demo –package=corp
```

where the demoin file includes the following:

```
SQL ADDRESS GENERATE JAddress AS MyAddress
```

JPublisher will generate these files:

```
demo/corp/JAddress.java
demo/corp/MyAddressRef.java
```

Because an ADDRESS object will be represented in the Java program as a MyAddress instance, JPublisher generates the class MyAddressRef rather than JAddressRef.

Here is a listing of the demo/corp/JAddress.java class file generated by JPublisher:

> **Note:** The details of method bodies that JPublisher generates might change in future releases.

```
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 JAddress implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  static int[] _sqlType =
  {
    12, 12, 12, 2
  };

  static ORADataFactory[] _factory = new ORADataFactory[4];

  static final JAddress _JAddressFactory = new JAddress();
  public static ORADataFactory getORADataFactory()
  {
    return _JAddressFactory;
  }

  /* constructor */
  protected JAddress(boolean init)
  { if(init) _struct = new MutableStruct(new Object[4], _sqlType, _factory); }
  public JAddress()
  { this(true); }

  /* 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(JAddress o, Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    if (o == null) o = new JAddress(false);
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
```

```
    return o;
  }

  /* accessor methods */
  public String getStreet() throws SQLException
  { return (String) _struct.getAttribute(0); }

  public void setStreet(String street) throws SQLException
  { _struct.setAttribute(0, street); }


  public String getCity() throws SQLException
  { return (String) _struct.getAttribute(1); }

  public void setCity(String city) throws SQLException
  { _struct.setAttribute(1, city); }

  public String getState() throws SQLException
  { return (String) _struct.getAttribute(2); }

  public void setState(String state) throws SQLException
  { _struct.setAttribute(2, state); }

  public java.math.BigDecimal getZip() throws SQLException
  { return (java.math.BigDecimal) _struct.getAttribute(3); }

  public void setZip(java.math.BigDecimal zip) throws SQLException
  { _struct.setAttribute(3, zip); }

}
```

Here is a listing of the `demo/corp/MyAddressRef.java` class file generated by
JPublisher:

```
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 MyAddressRef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;

  static final MyAddressRef _MyAddressRefFactory = new MyAddressRef();
  public static ORADataFactory getORADataFactory()
  {
    return _MyAddressRefFactory;
  }

  /* constructor */
  public MyAddressRef()
  {
  }

  /* 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;
    MyAddressRef r = new MyAddressRef();
    r._ref = (REF) d;
    return r;
  }

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

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

Example: Wrappers Generated for Methods in Objects

> **Note:** The wrapper methods that JPublisher generates to invoke stored procedures are generated in SQLJ code; therefore, JPublisher-generated classes that contain wrapper methods must be processed by the SQLJ translator.

This section describes an example of JPublisher output given the definition below of a SQL type containing methods. The example defines a type `Rational` with `numerator` and `denominator` attributes and the following functions and procedures:

- `MEMBER FUNCTION toReal`: Given two integers, this function converts a rational number to a real number and returns a real number.

- `MEMBER PROCEDURE normalize`: Given two integers, representing a numerator and a denominator, this procedure reduces a fraction by dividing the numerator and denominator by their greatest common divisor.

- `STATIC FUNCTION gcd`: Given two integers, this function returns their greatest common divisor.

- `MEMBER FUNCTION plus`: This function adds two rational numbers and returns the result.

The code for `rational.sql` follows:

```
CREATE TYPE Rational AS OBJECT (
 numerator INTEGER,
 denominator INTEGER,
 MAP MEMBER FUNCTION toReal RETURN REAL,
 MEMBER PROCEDURE normalize,
 STATIC FUNCTION gcd(x INTEGER,
                     y INTEGER) RETURN INTEGER,
 MEMBER FUNCTION plus ( x Rational) RETURN Rational
);

CREATE TYPE BODY Rational AS

MAP MEMBER FUNCTION toReal RETURN REAL IS
-- convert rational number to real number
BEGIN
  RETURN numerator / denominator;
END toReal;
```

```
MEMBER PROCEDURE normalize IS
 g INTEGER;
BEGIN
 g := Rational.gcd(numerator, denominator);
 numerator := numerator / g;
 denominator := denominator / g;
END normalize;

STATIC FUNCTION gcd(x INTEGER,
                    y INTEGER) RETURN INTEGER IS
-- find greatest common divisor of x and y
ans INTEGER;
z INTEGER;
BEGIN
IF x < y THEN
   ans := Rational.gcd(y, x);
ELSIF (x MOD y = 0) THEN
   ans := y;
ELSE
   z := x MOD y;
   ans := Rational.gcd(y, z);
END IF;
RETURN ans;
END gcd;

MEMBER FUNCTION plus (x Rational) RETURN Rational IS
BEGIN
 return Rational(numerator * x.denominator + x.numerator * denominator,
                 denominator * x.denominator);
END plus;
END;
```

In this example, JPublisher was invoked with the following command line:

```
jpub -user=scott/tiger -sql=Rational -methods=true
```

The -user parameter directs JPublisher to login to the database as user scott with password tiger. The -methods parameter directs JPublisher to generate wrappers for the methods contained in the type Rational. You can omit this parameter, because -methods=true is the default.

Listing and Description of Rational.sqlj Generated by JPublisher

JPublisher generates the file `Rational.sqlj`. This file reads as follows:

> **Notes:**
>
> - The details of method bodies that JPublisher generates might change in future releases.
>
> - Notice the `release()` calls, which are to avoid memory leaks related to SQLJ connection contexts. See "Releasing Connection Context Resources" on page 1-40 for more information.

```
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;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

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

  /* 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;
}

protected MutableStruct _struct;

static int[] _sqlType =
{
  4, 4
};

static ORADataFactory[] _factory = new ORADataFactory[2];

static final Rational _RationalFactory = new Rational(false);
public static ORADataFactory getORADataFactory()
{
  return _RationalFactory;
}

/* constructors */
protected Rational(boolean init)
{ if (init) _struct = new MutableStruct(new Object[2], _sqlType, _factory); }
public Rational()
{ this(true); __tx = DefaultContext.getDefaultContext(); }
public Rational(DefaultContext c) throws SQLException
{ this(true); __tx = c; }
public Rational(Connection c) throws SQLException
{ this(true); __onn = c; }

/* ORAData interface */
public Datum toDatum(Connection c) throws SQLException
{
  if (__tx!=null && __onn!=c) release();
  __onn = c;
  return _struct.toDatum(c, _SQL_NAME);
}

/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{ return create(null, d, sqlType); }
public void setFrom(Rational o) throws SQLException
{ release(); _struct = o._struct; __tx = o.__tx; __onn = o.__onn; }
protected void setValueFrom(Rational o) { _struct = o._struct; }
```

```
protected ORAData create(Rational o, Datum d, int sqlType) throws SQLException
{
  if (d == null) { if (o!=null) { o.release(); }; return null; }
  if (o == null) o = new Rational(false);
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o.__onn = ((STRUCT) d).getJavaSqlConnection();
  return o;
}

/* accessor methods */
public Integer getNumerator() throws SQLException
{ return (Integer) _struct.getAttribute(0); }

public void setNumerator(Integer numerator) throws SQLException
{ _struct.setAttribute(0, numerator); }


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

public void setDenominator(Integer denominator) throws SQLException
{ _struct.setAttribute(1, denominator); }


public Integer gcd (
  Integer x,
  Integer y)
throws SQLException
{
  Integer __jPt_result;
  #sql [getConnectionContext()] __jPt_result = { VALUES(RATIONAL.GCD(
    :x,
    :y)) };
  return __jPt_result;
}

public Rational normalize ()
throws SQLException
{
  Rational __jPt_temp = this;
  #sql [getConnectionContext()] {
    BEGIN
    :INOUT __jPt_temp.NORMALIZE();
    END;
  };
```

```
    return __jPt_temp;
  }

  public Rational plus (
    Rational x)
  throws SQLException
  {
    Rational __jPt_temp = this;
    Rational __jPt_result;
    #sql [getConnectionContext()] {
      BEGIN
      :OUT __jPt_result := :__jPt_temp.PLUS(
      :x);
      END;
    };
    return __jPt_result;
  }

  public Float toreal ()
  throws SQLException
  {
    Rational __jPt_temp = this;
    Float __jPt_result;
    #sql [getConnectionContext()] {
      BEGIN
      :OUT __jPt_result := :__jPt_temp.TOREAL();
      END;
    };
    return __jPt_result;
  }
}
```

All the methods that JPublisher generates invoke the corresponding PL/SQL methods executing in the server.

JPublisher declares the *sql_name* for the object to be SCOTT.RATIONAL and its *sql_type_code* to be OracleTypes.STRUCT. By default it uses the SQLJ connection context class sqlj.runtime.ref.DefaultContext. It creates accessor methods getNumerator(), setNumerator(), getDenominator(), and setDenominator() for the object attributes numerator and denominator.

JPublisher generates source code for the gcd static function, which takes two Integer values as input and returns an Integer result. This gcd function invokes the RATIONAL.GCD stored function with IN host variables :x and :y.

JPublisher generates source code for the `normalize` member procedure, which defines a PL/SQL block containing an `IN OUT` parameter inside the SQLJ statement. The `this` parameter passes the values to the PL/SQL block.

JPublisher generates source code for the `plus` member function, which takes an object `x` of type `Rational` and returns an object of type `Rational`. It defines a PL/SQL block inside the SQLJ statement. The `IN` host variables are `:x` and a copy of `this`. The result of the function is an `OUT` host variable.

JPublisher generates source code for the `toReal` member function, which returns a `Float` value. It defines a host `OUT` variable that is assigned the value returned by the function. A copy of the `this` object is an `IN` parameter.

Example: Wrappers Generated for Methods in Packages

> **Note:** The wrapper methods that JPublisher generates to invoke stored procedures are generated in SQLJ code; therefore, JPublisher-generated classes that contain wrapper methods must be processed by the SQLJ translator.

This section describes an example of JPublisher output given the definition below of a PL/SQL package containing methods. The example defines the package RationalP with the following functions and procedures, which manipulate the numerators and denominators of fractions.

- FUNCTION toReal: Given two integers, this function converts a rational number to a real number and returns a real number.

- PROCEDURE normalize: Given two integers (representing a numerator and a denominator), this procedure reduces a fraction by dividing the numerator and denominator by their greatest common divisor.

- FUNCTION gcd: Given two integers, this function returns their greatest common divisor.

- PROCEDURE plus: Adds two rational numbers and returns the result.

The code for RationalP.sql follows:

```
CREATE PACKAGE RationalP AS

 FUNCTION toReal(numerator    INTEGER,
                 denominator INTEGER) RETURN REAL;

 PROCEDURE normalize(numerator    IN OUT INTEGER,
                     denominator IN OUT INTEGER);

 FUNCTION gcd(x INTEGER, y INTEGER) RETURN INTEGER;

 PROCEDURE plus (n1 INTEGER, d1 INTEGER,
                 n2 INTEGER, d2 INTEGER,
                 n3 OUT INTEGER, d3 OUT INTEGER);
END rationalP;

/
```

```
CREATE PACKAGE BODY rationalP AS

  FUNCTION toReal(numerator INTEGER,
                  denominator INTEGER) RETURN real IS
  -- convert rational number to real number
  BEGIN
    RETURN numerator / denominator;
  END toReal;

  FUNCTION gcd(x INTEGER, y INTEGER) RETURN INTEGER IS
  -- find greatest common divisor of x and y
  ans INTEGER;
  BEGIN
  IF x < y THEN
      ans := gcd(y, x);
  ELSIF (x MOD y = 0) THEN
      ans := y;
  ELSE
      ans := gcd(y, x MOD y);
  END IF;
  RETURN ans;
  END gcd;

  PROCEDURE normalize( numerator IN OUT INTEGER,
                       denominator IN OUT INTEGER) IS
   g INTEGER;
   BEGIN
   g := gcd(numerator, denominator);
   numerator := numerator / g;
   denominator := denominator / g;
   END normalize;

  PROCEDURE plus (n1 INTEGER, d1 INTEGER,
                  n2 INTEGER, d2 INTEGER,
                  n3 OUT INTEGER, d3 OUT INTEGER) IS
  BEGIN
  n3 := n1 * d2 + n2 * d1;
  d3 := d1 * d2;
  END plus;

END rationalP;
```

In this example, JPublisher was invoked with the following command line:

```
jpub -user=scott/tiger -sql=RationalP -methods=true
```

The `-user` parameter directs JPublisher to login to the database as user `scott` with password `tiger`. The `-methods` parameter directs JPublisher to generate wrappers for the methods in the package `RationalP`. You can omit this parameter, because `-methods=true` is the default.

Listing and Description of RationalP.sqlj Generated by JPublisher

JPublisher generates the file `RationalP.sqlj`, which reads as follows:

> **Note:** The details of method bodies that JPublisher generates might change in future releases.

```
import java.sql.SQLException;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

public class RationalP
{

  /* 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 RationalP() throws SQLException
{ __tx = DefaultContext.getDefaultContext(); }
public RationalP(DefaultContext c) throws SQLException
{ __tx = c; }
public RationalP(Connection c) throws SQLException
{__onn = c; __tx = new DefaultContext(c); }

public Integer gcd (
  Integer x,
  Integer y)
throws SQLException
{
  Integer __jPt_result;
  #sql [getConnectionContext()] __jPt_result = { VALUES(RATIONALP.GCD(
    :x,
    :y)) };
  return __jPt_result;
}

public void normalize (
  Integer numerator[],
  Integer denominator[])
throws SQLException
{
  #sql [getConnectionContext()] { CALL RATIONALP.NORMALIZE(
    :INOUT (numerator[0]),
    :INOUT (denominator[0])) };
}

public void plus (
  Integer n1,
  Integer d1,
  Integer n2,
  Integer d2,
  Integer n3[],
  Integer d3[])
throws SQLException
{
  #sql [getConnectionContext()] { CALL RATIONALP.PLUS(
    :n1,
    :d1,
    :n2,
    :d2,
    :OUT (n3[0]),
    :OUT (d3[0])) };
```

```
  }

  public Float toreal (
    Integer numerator,
    Integer denominator)
  throws SQLException
  {
    Float __jPt_result;
    #sql [getConnectionContext()] __jPt_result = { VALUES(RATIONALP.TOREAL(
      :numerator,
      :denominator)) };
    return __jPt_result;
  }
}
```

All the methods that JPublisher generates invoke the corresponding PL/SQL methods executing in the server.

By default, JPublisher uses the existing SQLJ connection context class `sqlj.runtime.ref.DefaultContext` and associates an instance of it with the `RationalP` package.

JPublisher generates source code for the `gcd` function, which takes two `BigDecimal` values—`x` and `y`—and returns a `BigDecimal` result. This `gcd` function invokes the stored function `RATIONALP.GCD` with `IN` host variables `:x` and `:y`.

JPublisher generates source code for the `normalize` procedure, which takes two `BigDecimal` values—`numerator` and `denominator`. This normalize procedure invokes the stored procedure call `RATIONALP.NORMALIZE` with `IN OUT` host variables `:numerator` and `:denominator`. Because these are `IN OUT` parameters, JPublisher passes their values as the first element of an array.

JPublisher generates source code for the `plus` procedure, which takes four `BigDecimal IN` parameters and two `BigDecimal OUT` parameters. This `plus` procedure invokes the stored procedure call `RATIONALP.PLUS`, with `IN` host variables `:n1`, `:d1`, `:n2`, and `:d2`. It also defines the `OUT` host variables `:n3` and `:d3`. Because these are `OUT` variables, JPublisher passes each of their values as the first element of an array.

JPublisher generates source code for the `toReal` function, which takes two `BigDecimal` values—`numerator` and `denominator`—and returns a `BigDecimal` result. This `toReal` function invokes the stored function call `RATIONALP.TOREAL`, with `IN` host variables `:numerator` and `:denominator`.

Example: Using Classes Generated for Object Types

This section illustrates an example of how you can use the classes that JPublisher generates for object types. Suppose you have defined a SQL object type that contains attributes and methods. You use JPublisher to generate a `<name>.sqlj` file and a `<name>Ref.java` file for the object type. To enhance the functionality of the Java class generated by JPublisher, you can extend the class. After translating (if applicable) and compiling the classes, you can use them in a program. For more information on this topic, see "Use of Classes JPublisher Generates for Object Types" on page 1-35.

The following steps demonstrate the scenario described above. In this case, define a `RationalO` SQL object type that contains `numerator` and `denominator` attributes and several methods to manipulate rational numbers. After using JPublisher to generate the `JPubRationalO.sqlj` and a `RationalORef.java` files, provide a file, `RationalO.java`, that enhances the functionality of the `JPubRationalO` class by extending it. After compiling the necessary files, use the classes in a test file to test the performance of the `RationalO.java` class.

Here are the steps, followed by listings of the files:

1. Create the SQL object type `RationalO`. "Listing of RationalO.sql (Definition of Object Type)" on page 3-55 contains the code for the `RationalO.sql` file.

2. Use JPublisher to generate Java classes (a `JPubRationalO.sqlj` file and a `RationalORef.java` file) for the object. Use this command line:

```
jpub -props=RationalO.props
```

Assume the properties file `RationalO.props` contains the following:

```
jpub.user=scott/tiger
jpub.sql=RationalO:JPubRationalO:RationalO
jpub.methods=true
```

According to the properties file, JPublisher will log into the database with user name `scott` and password `tiger`. The `sql` parameter directs JPublisher to translate the object type `RationalO` (declared by `RationalO.sql`) and generate `JPubRationalO` as `RationalO`, where the second `RationalO` indicates a class that you have written (`RationalO.java`) that extends the functionality of the original `RationalO`. The value of the `methods` parameter indicates that JPublisher will generate classes for PL/SQL packages and wrapper methods.

JPublisher produces the following files:

```
JPubRationalO.sqlj
RationalORef.java
```

See "Listing of JPubRationalO.sqlj Generated by JPublisher" on page 3-56 and "Listing of RationalORef.java Generated by JPublisher" on page 3-60 for listings of the `JPubRationalO.sqlj` and `RationalORef.java` files.

3. Write a file `RationalO.java` that enhances the functionality of `JPubRationalO.sqlj` by extending it. In `RationalO.java`, everything is inherited from the superclass except the following items. Add code to do the following:

   - Declare a factory object, `_JPubRationalO`.

   - Implement a `getORADataFactory()` method.

   - Implement a `create()` method.

   - Implement the constructors by calling the constructors in the superclass.

   - Add a `toString()` method, which is used in the last two `System.out.println()` calls in the test program `TestRationalO.java` (described in "Listing of TestRationalO.java Written by User" on page 3-63).

   "Listing of RationalO.java Written by User" on page 3-61 contains the code for the `RationalO.java` file.

4. Use SQLJ to compile/translate the necessary files. Enter the following:

```
sqlj JPubRationalO.sqlj RationalO.java
```

   This translates and compiles `JPubRationalO.sqlj` and compiles the `RationalO.java` file.

5. Write a program `TestRationalO.java` that uses the `RationalO` class. "Listing of TestRationalO.java Written by User" on page 3-63 contains the code.

6. Create the file `connect.properties`, which `TestRationalO` uses to determine how to connect to the database. The file reads as follows:

```
sqlj.user=scott
sqlj.password=tiger
sqlj.url=jdbc:oracle:oci:@
sqlj.driver=oracle.jdbc.driver.OracleDriver
```

7. Compile and run `TestRationalO`:

```
javac TestRationalO.java
java TestRationalO
```

The program produces the following output:

```
gcd: 5
real value: 0.5
sum: 100/100
sum: 1/1
```

Listing of RationalO.sql (Definition of Object Type)

This section contains the code that defines the `RationalO` SQL object type.

```
CREATE TYPE RationalO AS OBJECT (
    numerator INTEGER,
    denominator INTEGER,
    MAP MEMBER FUNCTION toReal RETURN REAL,
    MEMBER PROCEDURE normalize,
    STATIC FUNCTION gcd(x INTEGER,
                        y INTEGER) RETURN INTEGER,
    MEMBER FUNCTION plus ( x RationalO) RETURN RationalO
);

CREATE TYPE BODY RationalO AS

  MAP MEMBER FUNCTION toReal RETURN REAL IS
  -- convert rational number to real number
  BEGIN
    RETURN numerator / denominator;
  END toReal;

  MEMBER PROCEDURE normalize IS
   g BINARY_INTEGER;
  BEGIN
   g := RationalO.gcd(numerator, denominator);
   numerator := numerator / g;
   denominator := denominator / g;
  END normalize;

  STATIC FUNCTION gcd(x INTEGER,
                      y INTEGER) RETURN INTEGER IS
  -- find greatest common divisor of x and y
  ans BINARY_INTEGER;
```

```
      BEGIN
      IF x < y THEN
         ans := RationalO.gcd(y, x);
      ELSIF (x MOD y = 0) THEN
         ans := y;
      ELSE
         ans := RationalO.gcd(y, x MOD y);
      END IF;
      RETURN ans;
      END gcd;

      MEMBER FUNCTION plus (x RationalO) RETURN RationalO IS
      BEGIN
       return RationalO(numerator * x.denominator + x.numerator * denominator,
                      denominator * x.denominator);
      END plus;
    END;
```

Listing of JPubRationalO.sqlj Generated by JPublisher

This section lists the code in JPubRationalO.java that JPublisher generates.

```
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;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

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

  /* 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;
}

protected MutableStruct _struct;

static int[] _sqlType =
{
  4, 4
};

static ORADataFactory[] _factory = new ORADataFactory[2];

static final JPubRationalO _JPubRationalOFactory = new JPubRationalO(false);
public static ORADataFactory getORADataFactory()
{
  return _JPubRationalOFactory;
}

/* constructors */
protected JPubRationalO(boolean init)
{ if (init) _struct = new MutableStruct(new Object[2], _sqlType, _factory); }
public JPubRationalO()
{ this(true); __tx = DefaultContext.getDefaultContext(); }
public JPubRationalO(DefaultContext c) throws SQLException
{ this(true); __tx = c; }
public JPubRationalO(Connection c) throws SQLException
{ this(true); __onn = c; }

/* ORAData interface */
public Datum toDatum(Connection c) throws SQLException
{
  if (__tx!=null && __onn!=c) release();
  __onn = c;
```

```
      return _struct.toDatum(c, _SQL_NAME);
    }

    /* ORADataFactory interface */
    public ORAData create(Datum d, int sqlType) throws SQLException
    { return create(null, d, sqlType); }
    public void setFrom(JPubRationalO o) throws SQLException
    { release(); _struct = o._struct; __tx = o.__tx; __onn = o.__onn; }
    protected void setValueFrom(JPubRationalO o) { _struct = o._struct; }
    protected ORAData create(JPubRationalO o, Datum d, int sqlType) throws
SQLException
    {
      if (d == null) { if (o!=null) { o.release(); }; return null; }
      if (o == null) o = new JPubRationalO(false);
      o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
      o.__onn = ((STRUCT) d).getJavaSqlConnection();
      return o;
    }

    /* accessor methods */
    public Integer getNumerator() throws SQLException
    { return (Integer) _struct.getAttribute(0); }

    public void setNumerator(Integer numerator) throws SQLException
    { _struct.setAttribute(0, numerator); }


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

    public void setDenominator(Integer denominator) throws SQLException
    { _struct.setAttribute(1, denominator); }


    public Integer gcd (
      Integer x,
      Integer y)
    throws SQLException
    {
      Integer __jPt_result;
      #sql [getConnectionContext()] __jPt_result = { VALUES(RATIONALO.GCD(
        :x,
        :y)) };
      return __jPt_result;
    }
```

```
public RationalO normalize ()
throws SQLException
{
  RationalO __jPt_temp = (RationalO) this;
  #sql [getConnectionContext()] {
    BEGIN
    :INOUT __jPt_temp.NORMALIZE();
    END;
  };
  return __jPt_temp;
}

public RationalO plus (
  RationalO x)
throws SQLException
{
  JPubRationalO __jPt_temp = this;
  RationalO __jPt_result;
  #sql [getConnectionContext()] {
    BEGIN
    :OUT __jPt_result := :__jPt_temp.PLUS(
    :x);
    END;
  };
  return __jPt_result;
}

public Float toreal ()
throws SQLException
{
  JPubRationalO __jPt_temp = this;
  Float __jPt_result;
  #sql [getConnectionContext()] {
    BEGIN
    :OUT __jPt_result := :__jPt_temp.TOREAL();
    END;
  };
  return __jPt_result;
}
}
```

Listing of RationalORef.java Generated by JPublisher

This section lists the code in `RationalORef.java` that JPublisher generates.

> **Note:** The details of method bodies that JPublisher generates might change in future releases.

```java
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 RationalORef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.RATIONALO";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;

  static final RationalORef _RationalORefFactory = new RationalORef();
  public static ORADataFactory getORADataFactory()
  {
    return _RationalORefFactory;
  }

  /* constructor */
  public RationalORef()
  {
  }

  /* 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;
```

```
      RationalORef r = new RationalORef();
      r._ref = (REF) d;
      return r;
   }

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

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

Listing of RationalO.java Written by User

This section lists the code for the user-written file, RationalO.java, that extends the class JPubRationalO.sqlj. Note that this program accomplishes the following:

- It declares a factory object, _JPubRationalO.

- It implements a getORADataFactory() method.

- It implements a create() method.

- It implements the constructors by calling the constructors in the superclass.

- It adds a toString() method, which is used in the last two System.out.println() calls in TestRationalO.java (described in "Listing of TestRationalO.java Written by User" on page 3-63).

Note: There is a way to code the ORADataFactory create() method to be more efficient than shown here, to ensure that an object instance is not needlessly created (if the data object is null) or needlessly re-initialized (if the data object is non-null). This is discussed in "Writing the Class that Extends the Generated Class" on page 1-45.

```java
import java.sql.SQLException;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;
import sqlj.runtime.ref.DefaultContext;
import java.sql.Connection;

public class RationalO extends JPubRationalO
                       implements ORAData, ORADataFactory
{
  /* _SQL_NAME inherited from JPubRationalO */

  /* _SQL_TYPECODE inherited from JPubRationalO */

  static final RationalO _RationalOFactory = new RationalO();
  public static ORADataFactory getORADataFactory()
  {
    return _RationalOFactory;
  }

  /* constructors */
  public RationalO()
  {
    super();
  }

  public RationalO(DefaultContext c) throws SQLException
  {
    super(c);
  }
  public RationalO(Connection c) throws SQLException
  {
    super(c);
  }

  /* ORAData interface */
  /* toDatum() inherited from JPubRationalO */

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    return create(new RationalO(), d, sqlType);
  }
```

```
/* accessor methods inherited from JPubRationalO */

/* additional method not in base class */
public String toString()
{
  try
  {
    return getNumerator().toString() + "/" + getDenominator().toString();
  }
  catch (SQLException e)
  {
    return null;
  }
}
}
```

Listing of TestRationalO.java Written by User

This section lists the contents of a user-written file, `TestRationalO.java`, that tests the performance of the `RationalO` class, given initial values for `numerator` and `denominator`. Note that the `TestRationalO.java` file also demonstrates how to do the following:

- Connect to the database by calling the `Oracle.connect()` method.

- Declare a Java object representing a SQL object type and initialize it by setting its attributes.

- Use the object to call server methods.

```
import oracle.sqlj.runtime.Oracle;
import oracle.sql.Datum;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Driver;

public class TestRationalO
{
  public static void main(String[] args)
  throws java.sql.SQLException
  {
    Oracle.connect(new TestRationalO().getClass(),
                   "connect.properties");
```

```
RationalO r = new RationalO();

Integer n = new Integer(5);
Integer d = new Integer(10);

r.setNumerator(n);
r.setDenominator(d);

Integer g = r.gcd(n, d);
System.out.println("gcd: " + g);

Float f = r.toreal();
System.out.println("real value: " + f);

RationalO s = r.plus(r);
System.out.println("sum: " + s);

s = s.normalize();
System.out.println("sum: " + s);
  }
}
```

Example: Using Classes Generated for Packages

This section provides an example of how you can use the classes and method wrappers that JPublisher generates for objects and packages, respectively. Suppose you have defined a SQL object type that contains attributes and a package with methods. You use JPublisher to generate a `<name>.sqlj` files for the object and the package. After translating the classes you can use them in a program. For more information on this topic, see "Use of SQLJ Classes JPublisher Generates for PL/SQL Packages" on page 1-34.

The following steps demonstrate the scenario described above. In this case, you define a `Rational` SQL object type that contains `numerator` and `denominator` integer attributes and a package `RationalP` that contains methods to manipulate rational numbers. After using JPublisher to generate the `Rational.sqlj` and `RationalP.sqlj` files, translate them with SQLJ, then use them in a test file to test the performance of the `Rational` and `RationalP` classes.

Here are the steps, followed by listings of the files:

1. Create the SQL object type `Rational` and package `RationalP`. "Listing of RationalP.sql (Definition of the Object Type and Package)" on page 3-66 contains the SQL code for the `RationalP.sql` file.

2. Use JPublisher to generate a Java class and a SQLJ class (`Rational.java` and `RationalP.sqlj`) for the object and package, respectively. Use this command line:

```
jpub -props=RationalP.props
```

Assume the properties file `RationalP.props` contains the following:

```
jpub.user=scott/tiger
jpub.sql=RationalP,Rational
jpub.mapping=oracle
jpub.methods=true
```

According to the properties file, JPublisher will log into the database with user name `scott` and password `tiger`. The `sql` parameter directs JPublisher to translate the object type `Rational` and package `RationalP` (declared in `RationalP.sql`). JPublisher will translate the type and package according to the `oracle` mapping. The value of the `methods` parameter indicates that JPublisher will generate classes for PL/SQL packages, including wrapper methods. Since the object type `Rational` does not have any member functions, JPublisher will translate it into a `.java` file, not a `.sqlj` file. By using the `-methods=always` setting for JPublisher, however, you could have requested

the generation of a `.sqlj` file regardless. See "Generate Classes for Packages and Wrapper Methods for Methods (-methods)" on page 2-17 for more information.

JPublisher produces the following files:

```
Rational.java
RationalP.sqlj
```

3. Translate the `RationalP.sqlj` and `Rational.java` files:

```
sqlj RationalP.sqlj Rational.java
```

4. Write a program, `TestRationalP.java`, that uses the `RationalP` class.

5. Write the file `connect.properties`, which `TestRationalP.java` uses to determine how to connect to the database. The file reads as follows:

```
sqlj.user=scott
sqlj.password=tiger
sqlj.url=jdbc:oracle:oci:@
sqlj.driver=oracle.jdbc.driver.OracleDriver
```

6. Compile and run `TestRationalP`:

```
javac TestRationalP.java
java TestRationalP
```

The program produces the following output:

```
gcd: 5
real value: 0.5
sum: 100/100
sum: 1/1
```

Listing of RationalP.sql (Definition of the Object Type and Package)

This section lists the contents of the file `RationalP.sql`, which defines the `Rational` SQL object type and the `RationalP` package.

```
CREATE TYPE Rational AS OBJECT (
   numerator INTEGER,
   denominator INTEGER
);
/
CREATE PACKAGE RationalP AS

 FUNCTION toReal(r Rational) RETURN REAL;
```

```
 PROCEDURE normalize(r IN OUT Rational);

 FUNCTION gcd(x INTEGER, y INTEGER) RETURN INTEGER;

 FUNCTION plus (r1 Rational, r2 Rational) RETURN Rational;

END rationalP;
/
CREATE PACKAGE BODY rationalP AS

  FUNCTION toReal(r Rational) RETURN real IS
  -- convert rational number to real number
  BEGIN
    RETURN r.numerator / r.denominator;
  END toReal;

  FUNCTION gcd(x INTEGER, y INTEGER) RETURN INTEGER IS
  -- find greatest common divisor of x and y
  result INTEGER;
  BEGIN
  IF x < y THEN
     result := gcd(y, x);
  ELSIF (x MOD y = 0) THEN
     result := y;
  ELSE
     result := gcd(y, x MOD y);
  END IF;
  RETURN result;
  END gcd;

  PROCEDURE normalize( r IN OUT Rational) IS
   g INTEGER;
   BEGIN
   g := gcd(r.numerator, r.denominator);
   r.numerator := r.numerator / g;
   r.denominator := r.denominator / g;
   END normalize;

  FUNCTION plus (r1 Rational,
                 r2 Rational) RETURN Rational IS
  n INTEGER;
  d INTEGER;
  result Rational;
  BEGIN
```

```
        n := r1.numerator * r2.denominator + r2.numerator * r1.denominator;
        d := r1.denominator * r2.denominator;
        result := Rational(n, d);
        RETURN result;
        END plus;

END rationalP;
/
```

Listing of TestRationalP.java Written by a User

The test program, TestRationalP.java, uses the package RationalP and the object type Rational, which does not have methods. The test program creates an instance of package RationalP and two Rational objects.

TestRationalP connects to the database in SQLJ style, using the Oracle.connect() method. In this example, the Oracle.connect() call specifies the file connect.properties, which contains these connection properties:

```
sqlj.url=jdbc:oracle:oci:@
sqlj.user=scott
sqlj.password=tiger
```

Following is a listing of TestRationalP.java:

```
import oracle.sql.Datum;
import oracle.sql.NUMBER;
import java.math.BigDecimal;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;
import java.sql.Connection;

public class TestRationalP
{

  public static void main(String[] args)
  throws java.sql.SQLException
  {

    Oracle.connect(new TestRationalP().getClass(),
                   "connect.properties");

    RationalP p = new RationalP();
```

```
            NUMBER n = new NUMBER(5);
            NUMBER d = new NUMBER(10);
            Rational r = new Rational();
            r.setNumerator(n);
            r.setDenominator(d);

            NUMBER f = p.toreal(r);
            System.out.println("real value: " + f.stringValue());

            NUMBER g = p.gcd(n, d);
            System.out.println("gcd: " + g.stringValue());

            Rational s = p.plus(r, r);
            System.out.println("sum: " + s.getNumerator().stringValue() +
                                "/" + s.getDenominator().stringValue());

            Rational[] sa = {s};
            p.normalize(sa);
            s = sa[0];
            System.out.println("sum: " + s.getNumerator().stringValue() +
                                "/" + s.getDenominator().stringValue());
    }
}
```

Example: Using Datatypes Not Supported by JDBC

One technique you can employ to use datatypes not supported by JDBC is to write an anonymous PL/SQL block that converts input types that JDBC supports into the input types that the PL/SQL method uses. Then convert the output types that the PL/SQL method uses into output types that JDBC supports. For more information on this topic, see "Using Datatypes Not Supported by JDBC" on page 1-29.

The following steps offer a general outline of how you would do this. The steps assume that you used JPublisher to translate an object type with methods that contain argument types not supported by JDBC. The steps describe the changes you must make. You could make changes by extending the class or modifying the generated files. Extending the classes is generally a better technique; however, in this example, the generated files are modified.

1. In Java, convert each `IN` or `IN OUT` argument having a type that JDBC does not support to a Java type that JDBC does support.

2. Pass each `IN` or `IN OUT` argument to a PL/SQL block.

3. In the PL/SQL block, convert each `IN` or `IN OUT` argument to the correct type for the PL/SQL method.

4. Call the PL/SQL method.

5. Convert each `OUT` argument or `IN OUT` argument or function result from the type that JDBC does not support to the corresponding type that JDBC does support in PL/SQL.

6. Return each `OUT` argument, `IN OUT` argument, or function result from the PL/SQL block.

7. In Java, convert each `OUT` argument, `IN OUT` argument, or function result from the type JDBC does support to the type it does not support.

Here is an example of how to handle an argument type not directly supported by JDBC. The example converts from/to a type that JDBC does not support (`Boolean/BOOLEAN`) to/from one that JDBC does support (`String/VARCHAR2`).

The following `.sql` file defines an object type with methods that use `boolean` arguments. The methods this program uses are very simple; they serve only to demonstrate that arguments are passed correctly.

```
CREATE TYPE BOOLEANS AS OBJECT (
 iIn     INTEGER,
 iInOut  INTEGER,
 iOut    INTEGER,
```

```
 MEMBER PROCEDURE p(i1 IN BOOLEAN,
                    i2 IN OUT BOOLEAN,
                    i3 OUT BOOLEAN),

 MEMBER FUNCTION f(i1 IN BOOLEAN) RETURN BOOLEAN
);

CREATE TYPE BODY BOOLEANS AS

MEMBER PROCEDURE p(i1 IN BOOLEAN,
                   i2 IN OUT BOOLEAN,
                   i3 OUT BOOLEAN) IS
BEGIN
  iOut := iIn;

  IF iInOut IS NULL THEN
    iInOut := 0;
  ELSIF iInOut = 0 THEN
    iInOut := 1;
  ELSE
    iInOut := NULL;
  END IF;

  i3 := i1;
  i2 := NOT i2;
END;

MEMBER FUNCTION f(i1 IN BOOLEAN) RETURN BOOLEAN IS
BEGIN
  return i1 = (iIn = 1);
END;

END;
```

The following `.sqlj` file was first generated by JPublisher and then modified by a
user, according to the steps above. The wrapper methods accomplish the following:

- Convert each argument from `Boolean` to `String` in Java.

- Pass each argument into a PL/SQL block.

- Convert the argument from `VARCHAR2` to `BOOLEAN` in PL/SQL.

- Call the PL/SQL method.

- Convert each `OUT` argument, `IN OUT` argument, or function result from `BOOLEAN` to `VARCHAR2` in PL/SQL.

- Return each `OUT` argument, `IN OUT` argument, or function result from the PL/SQL block.

- Finally, convert each `OUT` argument, `IN OUT` argument, or function result.

Here is the code:

```java
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;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

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

  /* 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;
  }
```

```
protected MutableStruct _struct;

static int[] _sqlType =
{
  4, 4, 4
};

static ORADataFactory[] _factory = new ORADataFactory[3];

static final Booleans _BooleansFactory = new Booleans(false);
public static ORADataFactory getORADataFactory()
{
  return _BooleansFactory;
}

/* constructors */
protected Booleans(boolean init)
{ if (init) _struct = new MutableStruct(new Object[3], _sqlType, _factory); }
public Booleans()
{ this(true); __tx = DefaultContext.getDefaultContext(); }
public Booleans(DefaultContext c) throws SQLException
{ this(true); __tx = c; }
public Booleans(Connection c) throws SQLException
{ this(true); __onn = c; }

/* ORAData interface */
public Datum toDatum(Connection c) throws SQLException
{
  if (__tx!=null && __onn!=c) release();
  __onn = c;
  return _struct.toDatum(c, _SQL_NAME);
}

/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{ return create(null, d, sqlType); }
public void setFrom(Booleans o) throws SQLException
{ release(); _struct = o._struct; __tx = o.__tx; __onn = o.__onn; }
protected void setValueFrom(Booleans o) { _struct = o._struct; }
protected ORAData create(Booleans o, Datum d, int sqlType) throws SQLException
{
  if (d == null) { if (o!=null) { o.release(); }; return null; }
   if (o == null) o = new Booleans(false);
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
```

```
    o.__onn = ((STRUCT) d).getJavaSqlConnection();
    return o;
}

/* accessor methods */
public Integer getIin() throws SQLException
{ return (Integer) _struct.getAttribute(0); }

public void setIin(Integer iin) throws SQLException
{ _struct.setAttribute(0, iin); }


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

public void setIinout(Integer iinout) throws SQLException
{ _struct.setAttribute(1, iinout); }


public Integer getIout() throws SQLException
{ return (Integer) _struct.getAttribute(2); }

public void setIout(Integer iout) throws SQLException
{ _struct.setAttribute(2, iout); }



/* Unable to generate method "f"
   because it uses a type that is not supported

public <unsupported type> f (
  <unsupported type> i1)
throws SQLException
{
  Booleans __jPt_temp = this;
  <unsupported type> __jPt_result;
  #sql [getConnectionContext()] {
    BEGIN
    :OUT __jPt_result := :__jPt_temp.F(
    :i1);
    END;
  };
  return __jPt_result;
} */
```

```
public Boolean f (
  Boolean i1)
throws SQLException
{
  Booleans _temp = this;
  String _i1 = null;
  String _result = null;

  if (i1 != null) _i1 = i1.toString();

  #sql [getConnectionContext()] {
    DECLARE
    i1_ BOOLEAN;
    result_ BOOLEAN;
    t_ VARCHAR2(5);

    BEGIN
    i1_ := :_i1 = 'true';

    result_ := :_temp.F(i1_);

    IF result_ THEN
      t_ := 'true';
    ELSIF NOT result_ THEN
      t_ := 'false';
    ELSE
      t_ := NULL;
    END IF;
    :OUT _result := t_;

    END;
  };

  if (_result == null)
      return null;
  else
      return new Boolean(_result.equals("true"));
}


/* Unable to generate method "p"
   because it uses a type that is not supported

 public Booleans p (
    <unsupported type> i1,
```

```
                     <unsupported type> i2[],
                     <unsupported type> i3[])
             throws SQLException
             {
               Booleans __jPt_temp = this;
               #sql [getConnectionContext()] {
                 BEGIN
                 :INOUT __jPt_temp.P(
                 :i1,
                 :INOUT (i2[0]),
                 :OUT (i3[0]));
                 END;
               };
               return __jPt_temp;
             } */

             public Booleans p (
               Boolean i1,
               Boolean i2[],
               Boolean i3[])
             throws SQLException
             {
               String _i1 = (i1 == null) ? null
                                         : i1.toString();

               String _i2 = (i2[0] == null) ? null
                                            : i2[0].toString();

               String _i3 = (i3[0] == null) ? null
                                            : i3[0].toString();

               Booleans _temp = this;

               #sql [getConnectionContext()] {
                 DECLARE
                 i1_ BOOLEAN;
                 i2_ BOOLEAN;
                 i3_ BOOLEAN;
                 t_ VARCHAR2(5);

                 BEGIN
                 i1_ := :_i1 = 'true';
                 i2_ := :_i2 = 'true';

                 :INOUT _temp.P( i1_, i2_, i3_);
```

```
        IF i2_ THEN
          t_ := 'true';
        ELSIF NOT i2_ THEN
          t_ := 'false';
        ELSE
          t_ := NULL;
        END IF;
        :OUT _i2 := t_;

        IF i3_ THEN
          t_ := 'true';
        ELSIF NOT i3_ THEN
          t_ := 'false';
        ELSE
          t_ := NULL;
        END IF;
        :OUT _i3 := t_;

        END;
      };

    i2[0] = (_i2 == null) ? null
                         : new Boolean(_i2.equals("true"));
    i3[0] = (_i3 == null) ? null
                         : new Boolean(_i3.equals("true"));
    return _temp;
  }
}
```

> **Note:** Because of the semantics of SQLJ parameters, it is necessary
> to assign to each output parameter exactly once within the block.

Index