# Oracle8*i*

JPublisher User's Guide

Release 2 (8.1.6)

December 1999

Part No. A81357-01

ORACLE

JPublisher User's Guide, Release 2 (8.1.6)

Part No. A81357-01



Primary Authors: P. Alan Thiesen, Thomas Pfaeffle

Contributor: Ellen Barnes, Prabha Krishna, Ekkehard Rohwedder

# Contents

## 2 JPublisher Examples

# Send Us Your Comments

**Oracle8*i* JPublisher User's Guide, Release 2 (8.1.6)**

**Part No. A81357-01**

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

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

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

- Electronic mail — jpgcomnt@us.oracle.com
- FAX - 650-506-7225. Attn: Java Platform Group, Information Development Manager
- Postal service:
  Oracle Corporation
  Information Development Manager
  500 Oracle Parkway, Mailstop 4op978
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

---

---

---

If you have problems with the software, please contact your local Oracle World Wide Support Center.

# Preface

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

This preface has the following sections:

## Intended Audience

This manual assumes that you are an experienced programmer and understand Oracle databases and the SQL, SQLJ, and Java programming languages. It also assumes that you understand the principles of JDBC.

## Manual Structure

This manual contains two chapters:

| | |
|---|---|
| Chapter 1, "JPublisher" | Introduces the JPublisher utility. This chapter also includes sections on the JPublisher command line, command line options, input files, and the datatype mappings used to translate object types and method arguments. |
| Chapter 2, "JPublisher Examples" | Presents examples of JPublisher output for object types and wrapper methods. |

## Related Documentation

This manual contains references to the following Oracle publications:

- *Oracle8i JDBC Developer's Guide and Reference*

  This book describes Oracle's implementation of the JDBC standard and the Oracle JDBC server-side and client-side drivers. The Oracle extensions provide mappings from SQL datatypes to Java classes and offer significant advantages in manipulating SQL data. The extensions also support the use of structured objects in the database.

- *Oracle8i SQLJ Developer's Guide and Reference*

  This book explains the use of SQLJ to embed static SQL operations directly into Java code. It describes both standard SQLJ features and Oracle-specific SQLJ features.

- *Oracle8i Java Stored Procedures Developer's Guide*

  This book covers Java stored procedures, which open the Oracle RDBMS to all Java programmers. With stored procedures (functions, procedures, database triggers, and SQL methods), Java developers can implement business logic at the server level, and thereby improve application performance, scalability, and security.

- *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*

  This manual discusses the Oracle extensions to the JavaBeans and CORBA specifications.

- *Oracle8i 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 (4GL), PL/SQL offers seamless SQL access, tight integration with the Oracle server and tools, portability, security, and modern software engineering features such as data encapsulation, overloading, exception handling, and information hiding. This guide explains all the concepts behind PL/SQL and illustrates every facet of the language.

## Conventions this Manual Uses

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.

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

Additionally, this manual uses the following conventions.

| Convention | Meaning |
|---|---|
| . . . | Vertical ellipsis points in an example mean that information not directly related to the example has been omitted. |
| . . . | Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted. |
| < > | Angle brackets enclose user-supplied names. |
| [ ] | Brackets enclose optional clauses from which you can choose one or none. |

# 1

# JPublisher

This chapter describes the following topics:

- Understanding JPublisher
- Using JPublisher
- Input Files
- Understanding Datatype Mappings
- Using SQLJ Classes JPublisher Generates
- Using Java Classes JPublisher Generates
- JPublisher Limitations

# Understanding JPublisher

This section contains these subsections:



## Introduction

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

- database object types
- database reference (REF) types
- database collection types (`varrays` or nested tables)
- PL/SQL packages

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

JPublisher generates `get` and `set` 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 the database.

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. Therefore, JPublisher generates `.sqlj` source files for classes with wrapper methods, that is, classes representing object types or PL/SQL packages. JPublisher generates `.java` source files for classes representing REF or collection types. If you ask JPublisher to

translate object types without generating wrapper methods, JPublisher will generate `.java` source files to represent the object types. You use SQLJ to compile the `.sqlj` files; you can use SQLJ or your Java compiler to compile the `.java` files.

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, REF, and collection types. The `oracle.sql` package contains simple generic classes that represent object, REF, 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 an Oracle8 server. 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.CustomDatum` interface or the `java.sql.SQLData` interface. Either interface makes it possible to transfer object type instances between the database and your Java program. For more information on the `CustomDatum` interface, see the *Oracle8i JDBC Developer's Guide and Reference.* See *The JDBC 2.0 API*, by Sun Microsystems, for more information on the `SQLData` interface.

**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.

## JPublisher Requirements

When you use version 8.1.6 of JPublisher, you should also use version 8.1.6 of SQLJ, because these two products are always installed together. To use all features of JPublisher, you also need:

- Oracle database version 8.1.6
- Oracle JDBC drivers version 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 1.1.1 or higher, rather than JDK 1.2.
- 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 `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 `CustomDatum` interface), you can use Oracle database version 8.1.5 with JDBC version 8.1.5 and JDK version 1.1.1 or higher.
- If you always tell JPublisher not to generate wrapper methods, you never generate any `.sqlj` files; therefore, you do not need to use SQLJ.

## 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 Net8 and required support files. If you do not have an Oracle client installation, JPublisher can use the Oracle Thin JDBC driver.

JPublisher generates a Java class for each database 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, you additionally must have the SQLJ runtime libraries.

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 `CustomDatum` classes (classes that implement the `oracle.sql.CustomDatum` interface) or `SQLData` classes (classes that implement the `java.sql.SQLData` interface).

The `CustomDatum` interface supports database object, REF, and collection types in a strongly typed way. That is, for each specific object, REF, or collection type in the database, there is a corresponding Java type. The `SQLData` interface, on the other hand, supports only database object types in a strongly-typed way. All REF types are represented generically as instances of `java.sql.Ref`, and all collection types are represented generically as instances of `java.sql.Array`. Therefore, JPublisher generates classes for REF and collection types only if it is generating `CustomDatum` classes.

When you run JPublisher for a user-defined object type and you request `CustomDatum` 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. This is necessary so that attributes can be materialized in Java whenever an instance of the top-level class is materialized.

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

When you run JPublisher for a user-defined collection type, you must request `CustomDatum` 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

  This is necessary so that 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 package's stored procedures

### Type Mappings

JPublisher maps the Oracle datatypes of the object's attributes and the PL/SQL package's arguments and function results to Java, according to mappings you specify globally. You can request Oracle-style or JDBC-style mappings. For numeric types, two other mapping styles are available: Object JDBC style and BigDecimal style.

- The JDBC mapping is based on the default Java mappings for SQL datatypes, as given in section 8.1 of the JDBC specification. It maps most numeric database types to Java primitive types such as `int` and `float`, and it maps DECIMAL and NUMBER to `java.math.BigDecimal`. Non-numeric predefined types map to standard Java classes such as `java.lang.String` and `java.sql.Blob`, and object types map to `SQLData` classes.
- The Object JDBC mapping affects only numeric types and is based on the mappings from SQL types to Java object types, as given in section 14.2 of the JDBC specification. It maps most numeric database types to Java wrapper classes such as `java.lang.Integer` and `java.lang.Float`, and it maps DECIMAL and NUMBER to `java.math.BigDecimal`. This mapping is the default for numeric types.
- The BigDecimal mapping affects only numeric types. As the name suggests, it maps all numeric database types to `java.math.BigDecimal`.
- The Oracle mapping maps Oracle datatypes to their corresponding `oracle.sql` classes. All numeric attribute types map to `oracle.sql.NUMBER`; non-numeric types map to Oracle types such as

`oracle.sql.CHAR` and `oracle.sql.BLOB`. User-defined types map to `CustomDatum` classes.

You can find a more detailed description of the Oracle, JDBC, and Object JDBC mappings in "Type Mappings for All Types (-mapping)" on page 1-26 and "Understanding Datatype Mappings" on page 1-46.

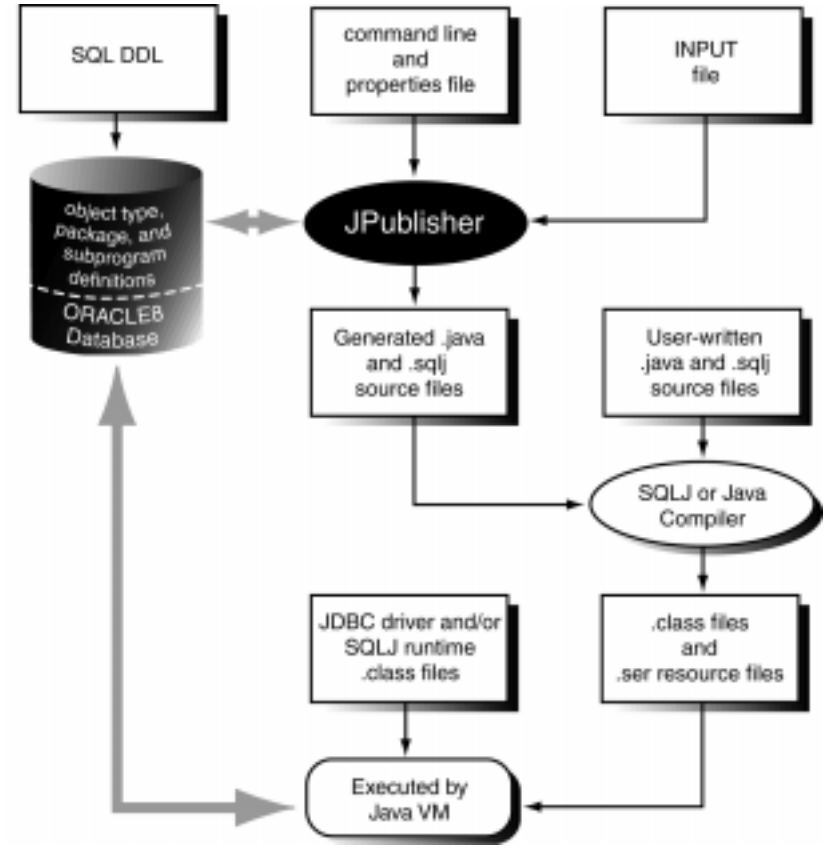
## Translating and Using PL/SQL Packages and Oracle Objects

Here are the basic steps for translating and using code for objects (including nested tables and `varrays`) and PL/SQL packages:

1. Create the desired object datatypes (Oracle objects) and PL/SQL packages in the database.
2. JPublisher generates source code for Java classes that represent PL/SQL packages, user-defined types, and `REF` types and places them in specified Java packages. JPublisher generates `.java` files for `REF`, `varray`, and nested table classes. If you ask JPublisher to generate wrapper methods, it will generate `.sqlj` files for object types and packages. If not, JPublisher will generate `.java` files without wrapper methods for object types and will not generate classes for packages (because they contain only wrapper methods).
3. Import these classes into your application code.
4. Use the methods in the generated classes to access and manipulate the Oracle objects and their attributes.
5. Compile all classes (the JPublisher-generated code and your code). The SQLJ compiler compiles the `.sqlj` files, and the Java or SQLJ 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 REF Types in Your Program

Here are the three ways in which you can represent user-defined object, collection, and REF types in your Java program:

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

  JPublisher generates classes for database object types that implement the SQLData interface. You can also write them by hand. If you write them by hand, your classes must then be registered using a type map.

  When you use the SQLData interface, all REF types are represented generically as instances of java.sql.Ref, and all collection types are represented generically as instances of java.sql.Array.

- 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.

- Use classes that implement the CustomDatum interface.

  JPublisher generates classes that implement the CustomDatum interface. You can also write them by hand.

Compared to classes that implement SQLData, classes that implement CustomDatum are fundamentally more efficient, because CustomDatum classes avoid unnecessary conversions to native Java types. For a comparison of the SQLData and CustomDatum interfaces, see the *Oracle8i JDBC Developer's Guide and Reference.*

Compared to oracle.sql.* classes, classes that implement CustomDatum 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 a CustomDatum that represents an Address.

JPublisher generated classes that implement CustomDatum or SQLData have additional advantages.

- The classes are customized, rather than generic. You access attributes of an object using get and set methods named after the particular attributes of the object.

- The classes are mutable. You can modify attributes of an object or elements of a collection. Exception: `CustomDatum` classes representing `REF` types are not mutable, because a `REF` does not have any subcomponents that could be sensibly modified.

## Sample JPublisher Command Line

On most operating systems, you invoke JPublisher on the command line. 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:

```
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 convenience, this chapter refers to the input file (the file specified by the `-input` option) as the `INPUT` file.

This command causes JPublisher to connect to the database with username `scott` and password `tiger` and translate database types to Java classes, based on instructions in the `INPUT` file `demoin`. The `-numbertypes=oracle` option directs JPublisher to map object attribute types to Java classes supplied by Oracle, and the `-usertypes=oracle` directs JPublisher to generate Oracle-specific `CustomDatum` classes. JPublisher places the classes that it generates in the package `corp` in the directory `demo`.

"JPublisher Options" on page 1-18 describes each of these options in more detail.

## Sample JPublisher Translation

This section illustrates a sample JPublisher translation of a simple 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 chapter.

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 command line:

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

Because `-dir=demo` was specified on the JPublisher command line, the package `corp` with the translated class `employee` is written to the file:

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

The `Employee.sqlj` class file would contain this code:

> **Note:** The details of the code JPublisher generates are subject to change in future releases. In particular, non-public methods, non-public fields, and all method bodies are subject to change.

```
package corp;

import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
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 Employee implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.EMPLOYEE";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  #sql static context _Ctx;
  _Ctx _ctx;

  MutableStruct _struct;

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

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

  static final Employee _EmployeeFactory = new Employee();
  public static CustomDatumFactory getFactory()
  {
    return _EmployeeFactory;
  }

  /* constructors */
  public Employee()
  {
    _struct = new MutableStruct(new Object[5], _sqlType, _factory);
    try
    {
      _ctx = new _Ctx(DefaultContext.getDefaultContext());
    }
    catch (Exception e)
    {
      _ctx = null;
    }
  }

  public Employee(ConnectionContext c) throws SQLException
  {
    _struct = new MutableStruct(new Object[5], _sqlType, _factory);
    _ctx = new _Ctx(c == null ? DefaultContext.getDefaultContext()
                              : c);
  }
```

```
public Employee(Connection c) throws SQLException
{
  _struct = new MutableStruct(new Object[5], _sqlType, _factory);
  _ctx = new _Ctx(c);
}

/* CustomDatum interface */
public Datum toDatum(OracleConnection c) throws SQLException
{
  _ctx = new _Ctx(c);
  return _struct.toDatum(c, _SQL_NAME);
}

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Employee o = new Employee();
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o._ctx = new _Ctx(((STRUCT) d).getConnection());
  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 also generates an `EmployeeRef.java` class. The contents of this file are displayed below.

```
package corp;

import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class EmployeeRef implements CustomDatum, CustomDatumFactory
{
  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 CustomDatumFactory getFactory()
  {
    return _EmployeeRefFactory;
  }

  /* constructor */
  public EmployeeRef()
  {
  }

  /* CustomDatum interface */
  public Datum toDatum(OracleConnection c) throws SQLException
```

```
  {
    return _ref;
  }

  /* CustomDatumFactory interface */
  public CustomDatum 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.getFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

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

You can find more examples of object mappings in "Example: JPublisher Type Mapping" on page 2-9.

# Using JPublisher

This section has the following subsections:



Before you run JPublisher, you must create any new database types that you will require. You must also ensure that any PL/SQL packages, methods, and subprograms that you want to invoke from Java are also installed in the database. Once you have done that, you can invoke JPublisher from the command line.

## Creating Types and Packages in the Database

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 REFs to object types. REF types are not explicitly declared in SQL. For more information on creating object types, see the *Oracle8i 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 advantages in these areas:

- 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.*

## Overview of JPublisher Input and Output

You can specify input options on the command line and in the properties file. In addition to producing `.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 1-18 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 1-38 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 1-37.

### 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) 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 and an `employeeRef.java` file.

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 `CustomDatum` 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.

## JPublisher Command Line Syntax

The JPublisher command-line syntax consists of the keyword `jpub`, followed by a list of JPublisher options. Consult your platform-specific documentation for instructions on how to invoke JPublisher on your platform.

Enter options on the JPublisher command line using this format:

*-option=value*

Your *option* entry is the literal option string and your *value* entry is a valid option setting. The literal option string is in lower case, and the *value* entry is case-sensitive. Separate command-line options with either spaces or tabs.

If you execute JPublisher without any options on the command line, it displays an option reference list and then terminates.

The following is an example of a JPublisher command line statement (enter it as one line, allowing it to wrap, as necessary):

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

**Notes:** No spaces are permitted around the equals sign (=).

The next section, "JPublisher Options", describes all the JPublisher options.

## JPublisher Options

Table 1–1 lists the options that you can use on the JPublisher command line, their syntax, and a brief description. Following the table, you can find a detailed description of each option. The abbreviation "n/a" represents "not applicable".

***Table 1–1 JPublisher Option Quick Guide***

| Option Name | Description | Default Value |
|---|---|---|
| `-builtintypes` | Specifies the type mappings (`jdbc` or `oracle`) for non-numeric, non-LOB built-in database types. | `jdbc` |
| `-case` | Specifies the case of Java identifiers that JPublisher generates. | `mixed` |
| `-dir` | Specifies the directory that holds generated packages. | current directory |
| `-driver` | Specifies the JDBC driver JPublisher uses to connect to the database. | `oracle.jdbc.driver.OracleDriver` |
| `-encoding` | Specifies the Java encoding of JPublisher's 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 |

***Table 1–1 JPublisher Option Quick Guide (Cont.)***

| Option Name | Description | Default Value |
|---|---|---|
| `-lobtypes` | Specifies the type 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. | `objectjdbc` |
| `-methods` | Determines whether JPublisher generates classes for PL/SQL packages and wrapper methods for methods in packages and object types, and whether JPublisher generates all methods or just the ones the user specifies. | `all` |
| `-numbertypes` | Specifies the type mappings (`jdbc`, `objectjdbc`, `bigdecimal`, or `oracle`) JPublisher uses for numeric database types. | `objectjdbc` |
| `-omit_schema_names` | Specifies whether all object type and package names JPublisher generates include the schema name. | do not omit schema names |
| `-package` | Specifies the name of the Java package for which JPublisher is generating a Java wrapper. | 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: The* `-types` *option is currently supported for compatibility, but deprecated in favor of* `-sql`. | n/a |

*Table 1–1 JPublisher Option Quick Guide (Cont.)*

| Option Name | Description | Default Value |
|---|---|---|
| `-url` | Specifies the URL JPublisher uses to connect to the database. | `jdbc:oracle:oci8:@` |
| `-user` | Specifies an Oracle username and password. | n/a |
| `-usertypes` | Specifies the types mappings (`jdbc` or `oracle`) JPublisher uses for used-defined database types. | `oracle` |

### JPublisher Option Tips

- 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 occur. Options from an INPUT file are processed at the point where the `-input` option occurs.
- If a particular option appears more than once, JPublisher uses the option's value from its last occurrence.
- We recommend that you 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 -types=Person -package=e.f ...
  ```

  or in the properties file you could enter:

  ```
  jpub.types=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 1-37 describes the properties file.

- On the command line, if you do not specify a type or package in the `INPUT` file or with the `-sql` or `-types` option, then JPublisher translates all types and packages in the user's schema according to the options you specify 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.
- You enter terms in UPPERCASE as shown, except that case is not significant.
- Terms in italics are like variables: you must specify an actual value or string.
- Square brackets [...] enclose optional items.
- Braces {...} enclose a list of possible values; you 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 so on.

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

### Options That Affect Type Mappings

The following options control which type mappings JPublisher uses to translate object types, collection types, REF types, and PL/SQL packages to Java classes:

- The `-builtintypes` option controls type mappings for non-numeric, non-lob, predefined SQL and PL/SQL types.
- The `-lobtypes` option controls type mappings for the BLOB and CLOB types.
- The `-numbertypes` option controls type mappings for numeric types.
- The `-usertypes` option controls JPublisher's behavior for user-defined types.

These four options are known as the type mapping options. (Another, less flexible option, `-mapping`, which is supported for compatibility with older releases of JPublisher, is discussed later.)

For an object type, JPublisher applies the mappings specified by the type mapping options to the object's 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 `get` methods return and the `set` 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 generates CustomDatum classes or `SQLData` classes and whether JPublisher generates code for collection and REF types.

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

Next, we discuss the preceding four possibilities.

**JDBC Mapping** The JDBC mapping maps most numeric database types 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 data types 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 database 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`.

**Object JDBC Mapping** The Object JDBC mapping maps most numeric database types 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 database 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** The BigDecimal mapping, as the name implies, maps all numeric database types 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 REF types, JPublisher generates `CustomDatum` classes.

Because the Oracle mapping uses no primitive types, it can represent a database 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.

#### More Information About Type Mappings

For an example of JPublisher output when `-mapping=objectjdbc` and when `-mapping=jdbc`, see "JPublisher Translation with the Oracle mapping" on page 2-5 and "" on page 2-8.

See "Understanding Datatype Mappings" on page 1-46 for more information about these mappings and for specific Oracle, JDBC, and Object JDBC type mappings.

See "Understanding Datatype Mappings" on page 1-46 for information about the factors you should consider when you decide which type mappings to use.

The next five sections discuss each of the type mapping options in detail.

#### Mappings For Built-In Types (-builtintypes)

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

The `-builtintypes` option controls type mappings for all the built-in database types except the LOB and BLOB types (controlled by the `-lobtypes` option) and the different numeric types (controlled by the `-numbertypes` option). The following table lists the database types affected by the `-builtintypes` option, and shows their Java type mappings for `-builtintypes=oracle` and for `-builtintypes=jdbc` (the default).

| SQL or PL/SQL Datatype | Oracle Mapping Class | JDBC Mapping Class |
|---|---|---|
| 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 LOB Types (-lobtypes)

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

The `-lobtypes` option controls type mappings for the LOB and BLOB types. The following table shows how these types are mapped for `-lobtypes=oracle` (the default) and for `-lobtypes=jdbc`.

| SQL and PL/SQL Datatype | Oracle Mapping Class | Object JDBC Mapping Class |
|---|---|---|
| CLOB | `oracle.sql.CLOB` | `java.sql.CLOB` |
| BLOB | `oracle.sql.BLOB` | `java.sql.BLOB` |

The BFILE type does not appear in this table, because it has only one mapping. It is always mapped to `oracle.sql.BFILE`, because there is no `java.sql.BFILE` class.

The `java.sql.CLOB` and `java.sql.BLOB` classes are new in JDK 1.2. If you use JDK 1.1, you should not select `-lobtypes=jdbc`.

#### Mappings For Numeric Types (-numbertypes)

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

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

- The JDBC mapping maps most numeric database types 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 database types 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 database types to `java.math.BigDecimal`.
- The Oracle mapping maps all numeric database types to `oracle.sql.NUMBER`.

The following table 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).

| SQL and PL/SQL Datatype | JDBC Mapping Class | Object JDBC Mapping Class |
|---|---|---|
| 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 |
| DEC, DECIMAL, NUMBER, NUMERIC | java.math.BigDecimal | java.math.BigDecimal |

#### Mappings for User-Defined Types (-usertypes)

`-usertypes={oracle|jdbc}`

The `-usertypes` option controls whether JPublisher generates `CustomDatum` classes or `SQLData` classes for user-defined types.

When `-usertypes=oracle` (the default), JPublisher generates `CustomDatum` classes for object, collection, and REF types.

When `-usertypes=jdbc`, JPublisher generates `SQLData` classes for object types. JPublisher does not generate anything for collection or REF types. Use `java.sql.Array` for all collection types, and `java.sql.Ref` for all REF types.

#### Type Mappings for All Types (-mapping)

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

You can use the `-mapping` option as an abbreviation for several common combinations of the type mapping options. This option is less flexible than setting the type mapping options individually and is supported mainly for compatibility with earlier versions of JPublisher.

`-mapping=oracle` is equivalent to setting all the type mapping options to `oracle`. The other mapping choices are equivalent to setting `-numbertypes` equal to the value of `-mapping` and setting the other type mapping options to their defaults, as shown in the following table:

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

If you do not specify a value for the `-mapping` option on the command line, then JPublisher assumes that `-mapping=objectjdbc`.

Because options are processed in the order in which they occur, if the `-mapping` option precedes one of the type mapping options (`-builtintypes`, `-lobtypes`, `-numbertypes`, or `-usertypes`) the type mapping option overrides the `-mapping` option for the types the type mapping option affects. If the `-mapping` option follows one of the type mapping options, the type mapping option is ignored.

### Other Options

Next, we discuss the remaining options in alphabetical order.

#### Case of Java Identifiers (-case)

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

The `-case` option affects the case of certain Java identifiers that JPublisher generates, including class names, method names, attribute names embedded within `get` and `set` method names, arguments of generated method names, and Java wrapper names. For class or attribute names that you enter with the `-sql` or `-types` option, or class names in the INPUT file, JPublisher retains the case of the letters in the name, overriding the `-case` option.

Table 1–2 describes the possible values for the `-case` option.

***Table 1–2 Values for the -case Option***

| -case Option Value | Description |
|---|---|
| `mixed` (default) | The first letter of every word of a class name or every word after the first word of a method name is in uppercase. All other characters are in lower case. An underscore (_), a dollar sign ($), and any character that is not a Java letter or digit constitutes a word boundary and is silently removed. A word 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 not a Java letter or digit 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 not a Java letter or digit 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 not a Java letter or digit and issues a warning message. |

If you do not enter a value for `-case` on the command line, JPublisher assumes that `-case=mixed`.

The `-case` option affects only those identifiers (attributes or types) that the `INPUT` file or `-sql` option does not explicitly mention. JPublisher performs case conversion after it generates a legal identifier.

**Case of Java Class Identifiers** 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 ... ;
```

### Output Directory for Generated Files (-dir)

`-dir=`*<directory name>*

The `-dir` option 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.

JPublisher combines the directory specified by `-dir`, 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` file. The "Name for Generated Packages (-package)" section on page 1-31 discusses this in more detail.

For example, in the following command line:

```
jpub -user=scott/tiger -input=demoin -mapping=oracle -case=lower -types=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 the current directory.

### JDBC Driver for Database Connection (-driver)

`-driver=`*<driver_name>*

The `-driver` option specifies the JDBC driver that JPublisher uses when it connects to the database. The default is:

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

This string, `oracle.jdbc.driver.OracleDriver`, 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, which represents ISO Latin-1.

As a general rule, you need not 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 by going to the URL http://www.javasoft.com and searching for "native-to-ascii".

### 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 1-41.

If you do not specify an `INPUT` file, or a package or object type with `-sql` or `-types` 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 1-38.

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

`-methods=(true|all|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`, JPublisher generates wrapper methods for all the methods in the object types and PL/SQL packages it processes. 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 at all, 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.

`CustomDatum` and SQLData classes generated by JPublisher include a static final String that names the database 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` file. The directory structure also reflects the package name. An explicit name in the `INPUT` file, after the `-sql` or `-types` option, overrides the value given to the `-package` option.

**Example** If `-dir=/a/b -package=c.d -case=mixed` appears on the command line, and the `INPUT` file contains the line:

```
SQL PERSON AS Person
```

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

```
-sql=PERSON:Person
-types=PERSON:Person
SQL PERSON AS Person
TYPE PERSON AS Person
-sql=PERSON
-types=PERSON
SQL PERSON
TYPE PERSON
```

The `Person.java` file contains (among other things) the package declaration "`package c.d;`".

Given the same command line and `INPUT` file contents as above, JPublisher creates the file `a/b/e/f/Person.java` in the following cases:

```
-sql=PERSON=e.f.Person;
-types=PERSON:e.f.Person;
SQL PERSON AS e.f.Person;
TYPE PERSON AS e.f.Person;
```

The `Person.java` file contains (among other things) the package declaration "`package e.f;`".

If you do not supply a package name for a class by any of the methods 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 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.

---

**Note:** The `-props` option is not allowed in the properties file.

---

If you do not specify a properties file, then JPublisher will use the processing commands entered on the command line.

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 1-37.

### Declare 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.

This section explains the -sql option syntax by translating it to the equivalent INPUT file syntax. "INPUT File Structure and Syntax" on page 1-38 explains the INPUT file syntax.

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. For more information on the `toplevel` keyword, see "Translating Top Level PL/SQL Subprograms" on page 1-34.

If you do not enter any types or packages to translate in the `INPUT` file or with the `-sql` or `-types` options, then JPublisher will translate all the types and packages in the schema to which you are connected.

In this section, we explain the `-sql` option by translating it to the equivalent INPUT file syntax. INPUT file syntax is explained in "Understanding the Translation Statement" on page 1-38.

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.

  ---

  **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:MyAddress:JAddress
```

JPublisher will interpret this as:

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

JPublisher

- creates a wrapper for the `CORPORATION` package
- translates the object type `Employee` as `oracleEmployee`
- translates `ADDRESS` as `JAddress`, generating code so that ADDRESS objects will be represented by the `MyAddress` class that you will write to extend `JAddress`
- and creates the references to the `MyAddress` class that you will write to extend `JAddress`.

**Translating Top Level PL/SQL Subprograms** 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*.

  For more information on the `TYPE`, `TYPE...AS`, and `TYPE...GENERATE...AS` syntax, see "Understanding the Translation Statement" on page 1-38.

You 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:MyAddress:JAddress
```

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:oci8:@
```

where you can follow the @ symbol by the 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.

### 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
```

# Input Files

These sections describe the structure and contents of JPublisher's 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 prefix "`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 from left to right, 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 (right-most) value.

For example, the options in this command line:

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

can be represented on the command line as:

```
jpub -props=my_properties
```

In this case, the contents of the properties file `my_properties` would be:

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

> **Note:** You must include the "`jpub.`" prefix 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 1-18 describes all the JPublisher options.

## INPUT File Structure and Syntax

You 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 an `INPUT` file or specify individual types or packages on the command line with the `-sql` (or `-types`) option, 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, you can enter TYPE <*type_name*> if you are specifying only object types.

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

You can also 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*>.

**GENERATE <*java_name_1*>] Clause** This clause specifies the name of the class that JPublisher generates. You use the GENERATE clause in conjunction with the AS clause. The AS clause specifies the name of the class that your Java program will use to represent the database object type. When the GENERATE clause does not appear, JPublisher generates this class (the class in the AS clause). But when the GENERATE clause is used, the user writes the class in the AS clause, and JPublisher generates the class in the GENERATE clause.

You use the GENERATE clause when you want to extend the class generated by JPublisher and map the database object type to the subclass.

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. For more information on this clause see "Extending JPublisher Classes" on page 1-43.

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 database 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.

**[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. It can include a package identifier. The Java name's case 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 1-41.

When you use the AS clause in conjunction with the GENERATE clause, the *java_name_2* is the name of the class that the user writes. For more information, see "GENERATE <java_name_1>] Clause" on page 1-39 and "Extending JPublisher Classes" on page 1-43.

**[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.
- The programmer prefers a different name.

Remember that your attribute names will appear embedded within `get` and `set` 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.

**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 1-31.

**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 1-10. If 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
);
```

then 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, we take the example in "Sample JPublisher Translation" on page 1-10 and make it more complex. For the command 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 an object type in the database 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 file:

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

This indicates that the Java source file Employee.java 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.

### Extending JPublisher Classes

You might want to enhance the functionality of a custom Java class generated by JPublisher by adding methods and transient fields to it. 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, and then map the object type to the subclass.

For example, 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, and 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 REF class MyAddressRef, rather than JAddressRef.
- JPublisher will use the MyAddress class, instead of the JAddress class, to represent attributes whose database type is ADDRESS.
- JPublisher will use the MyAddress class, instead of the JAddress class, to represent varray and nested table elements whose database type is ADDRESS.

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

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
```

If you were to enter the line in the INPUT file, instead of on the command line, it would look like this:

```
SQL ADDRESS GENERATE JAddress AS MyAddress
```

In this syntax, the 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 from the database that has an ADDRESS attribute, then 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 2-28.

**Writing the Class that Extends the Generated Class** The class that you create (for example, MyAddress.java) must have the following features. "Example: Generating a SQLData Class" on page 2-28 illustrates all these 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 CustomDatum interface or the SQLData interface. The simplest way to do this is to inherit the necessary methods from the superclass.
- If you are extending a CustomDatum class, you must also implement the CustomDatumFactory interface, either in the same class or in a different one. For example, you could have a class Employee that implements CustomDatum and a class EmployeeFactory that implements CustomDatumFactory.

**Adding Methods to JPublisher-generated Classes** Another way to enhance the functionality of a JPublisher-generated class is to add methods to it. However, we do not recommend adding methods to the generated class 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.

## INPUT File Precautions

This section describes some of the common errors it is possible to make in the `INPUT` file. 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 the file 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 `get` and `set` methods for both attributes without issuing a warning message. The question of whether the generated class is valid in Java depends on whether the two `get` methods with the same name and the two `set` 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.

# Understanding Datatype Mappings

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

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

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

The class that JPublisher generates for an object type will have `get` and `set` methods for the object's attributes. The class that JPublisher generates for a `varray` or nested table type will have `get` and `set` 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 mapping maps most numeric database types to standard Java types, including the primitive types `short`, `int`, `float`, and `double`. These types do not protect against null data, floating-point loss of precision, or integer overflow. This might not be a problem, however, if your object types are populated with non-null Java data.
- The Object JDBC mapping affects only numeric types. It is like the JDBC mapping except that JPublisher uses wrapper classes in the `java.lang` package, instead of Java primitive types. For example, JPublisher uses `java.lang.Integer` and `java.lang.Float`, instead of `int` and `float`. This allows null values, but does not protect against loss of precision or integer overflow. This is the default mapping for numeric types.
- The BigDecimal mapping, as the name suggests, maps all numeric database types to `java.math.BigDecimal`. The `BigDecimal` class supports null values and very large numbers, but conversions to and from `BigDecimal` are rather slow. This mapping affects only numeric types.
- The Oracle mapping maps Oracle datatypes to their corresponding `oracle.sql` classes. All numeric attribute types map to `oracle.sql.NUMBER`, which supports null values and very large numbers.

Non-numeric types map to Oracle types such as `oracle.sql.CHAR` and `oracle.sql.BLOB`, and user-defined types map to `CustomDatum` classes.

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 in the database 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 database types 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`s and `INSERT`s 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 1–3 lists the mappings from SQL and PL/SQL datatypes to Java types using the Oracle and JDBC mappings. You can use all the datatypes listed as supported in Table 1–3 as argument or result types for PL/SQL methods. You can use a subset of the datatypes as object attribute types. Table 1–4 on page 1-49 contains a list of these datatypes.

In Table 1–3, the **SQL and PL/SQL Datatype** column contains all possible datatypes.

The **Oracle Mapping Class** column contains 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 database types to Java types.

The **JDBC Mapping Class** column contains 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 *Oracle8i JDBC Developer's Guide and Reference* for more information on the `oracle.sql.*` package.

A few PL/SQL datatypes are not currently supported by JPublisher, because they are not currently supported by the Oracle JDBC drivers. These are designated in the table by the phrase "*not directly supported by JDBC*".

***Table 1–3 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 directly supported by JDBC* | *not directly supported by JDBC* |
| 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 |
| 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 directly supported by JDBC* | *not directly supported by JDBC* |
| object type | generated class | generated class |
| 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 |
| REF to object type | generated class implemented using oracle.sql.REF | java.sql.Ref |

*Table 1–3 PL/SQL Datatype to Oracle and Object JDBC Mapping Classes (Cont.)*

| SQL and PL/SQL Datatype | Oracle Mapping | JDBC Mapping |
|---|---|---|
| REF CURSOR | java.sql.ResultSet | java.sql.ResultSet |
| index-by tables | *not directly supported by JDBC* | *not directly supported by JDBC* |
| 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)".

## Allowed Object Attribute Types

You can use a subset of the PL/SQL datatypes listed in Table 1–3 as object attribute types. These datatypes are contained in Table 1–4, and have the same Oracle, Object JDBC, and JDBC mappings as described in Table 1–3 and Table 1–4.

*Table 1–4 Datatypes That Can be Used as Object Attribute Types*

| |
|---|
| 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 |
| REF to object 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 data types (such as NCHAR or 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 2-70.

## 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 crated objects. These new values must be returned to the wrapper method's 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 in the previous section 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

Similar problems 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 two different ways:

- If the stored procedure does not return a result (in PL/SQL terminology, it is a procedure rather than a function) the new value of `this` is returned as the function result of the wrapper method. This is the usual case, because the default mode for `this` in a stored procedure that returns a result is IN.
- If the stored procedure returns a result (in PL/SQL terminology, it is a function), the wrapper method returns the result of the stored procedure as its result. The new value of `this` is returned in a single-element array, passed as an extra argument to the wrapper method and is the last argument of the wrapper method.

## 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 translated to Java with `-mapping=oracle`, 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 settings `-methods=true` and `-mapping=oracle`. 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.

# Using SQLJ Classes JPublisher Generates

When `-methods=true`, JPublisher generates `.sqlj` classes for object types and PL/SQL packages. 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.

## Using 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 class's wrapper methods.

The constructors for the class associate a connection with the class. One constructor takes a `ConnectionContext`, one constructor takes a `Connection`, and one constructor has no arguments. Calling the no-argument constructor is equivalent to passing the `DefaultContext` to the constructor that takes a `ConnectionContext`. We supply the constructor that takes a `Connection`, for the convenience of the JDBC programmer who knows how to compile a SQLJ program, but is unfamiliar with SQLJ concepts such as `ConnectionContext`.

The wrapper methods are all instance methods, because the `ConnectionContext` 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 *Oracle8i SQLJ Developer's Guide and Reference* discusses reasons for using more than one connection context.

An instance of a class generated for a PL/SQL package does not contain copies of PL/SQL package variables. It is not a `CustomDatum` class or a `SQLData` class, and you cannot use it as a host variable.

"Example: Using Classes Generated for Packages" on page 2-65 shows how to use a class generated for a PL/SQL package.

## Using Classes JPublisher Generates for Object Types

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

To initialize your object, you can:

- Assign an already initialized object to your object.
- Retrieve a copy of a database object into your object. To do this, you can:
  - Use the object as an `OUT` argument or as the function call result of a JPublisher-generated wrapper method.
  - Retrieve the object through `#sql` statements you write.
  - Retrieve the object through JDBC calls you write.
- Construct the object and set its attributes using the `set` methods.

  The constructors for the class associate a connection with the class. One constructor takes a `ConnectionContext`, one constructor takes a `Connection`, and one constructor has no arguments. Calling the no-argument constructor is equivalent to passing the `DefaultContext` to the constructor that takes a `ConnectionContext`. We provide the constructor that takes a `Connection` for the convenience of the JDBC programmer who knows how to compile a SQLJ program, but is unfamiliar with SQLJ concepts such as `ConnectionContext`.

Once you have initialized your object, you can:

- Call the object's accessor methods.
- Call the object's wrapper methods.
- Pass the object to other wrapper methods.
- Use the object as a host variable in your own `#sql` statements.
- Use the object as a host variable in your own JDBC calls.

Logically speaking, there is a Java attribute for each attribute of the corresponding database object type. The object has `get` and `set` 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 `ConnectionContext` in the `this` object is used in `#sql` statements in the wrapper methods.

The following methods that JPublisher generates for the use of Oracle's JDBC drivers, are not intended for your use:

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

The `getFactory()`, `create()`, and `toDatum()` methods help implement the `CustomDatum` and `CustomDatumFactory` interfaces, which the Oracle JDBC drivers use and Oracle SQLJ requires.

The `RationalO` example, described in "Example: Using Classes Generated for Object Types" on page 2-53, shows how to use a class that was generated for an object type and has wrapper methods.

## General Comments about SQLJ Code JPublisher Generates

Every `.sqlj` class JPublisher generates for an object type or package has its own connection context class and its own connection context instance, declared within the class as:

```
#sql context _Ctx;
_Ctx ctx;
```

Having different connection context classes in different generated classes gives you the option of checking different classes against different example schemas during SQLJ on-line semantics checking.

Having different connection context instances in different objects guarantees that your objects can be used in different Java threads without explicit synchronization.

Your SQLJ code can use `.java` classes generated by JPublisher for `REF`, `varray`, and nested table types, and for object types generated by JPublisher when `-methods=false`. Having no embedded `#sql` statements, these are ordinary `.java` files.

You can use instances of these classes in your own `#sql` statements as host variables, and as arguments to wrapper methods generated by JPublisher. The next section further discusses them.

### Using Java Classes JPublisher Generates

When `-methods=false`, JPublisher does not generate wrapper methods for object types, and it does not generate code for PL/SQL packages at all, because they are

not useful without wrapper methods. When `-methods=false`, JPublisher exclusively generates `.java` files.

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

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

To initialize your object, you can:

- Assign an already initialized object to your object.
- Retrieve a copy of a database object into your object. To do this, you can:
  - Use the object as an `OUT` argument or as the function call result of a JPublisher-generated wrapper method in some other class.
  - Retrieve the object through `#sql` statements you write.
  - Retrieve the object through JDBC calls you write.
- Construct the object and initialize its data.

Unlike the constructors generated in `.sqlj` classes, the constructors generated in `.java` classes do not take a connection argument. Instead, when your object is passed to or returned from a `Statement`, `CallableStatement`, or `PreparedStatement`, JPublisher applies the connection it uses to construct the `Statement`, `CallableStatement`, or `PreparedStatement`.

This does not mean that 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 `REF` or a `BLOB`, that is valid only for a particular connection.

To initialize the object's data, you use the `set` methods if your class represents an object type, and you use the `setArray()` or `setElement()` methods if your class represents a `varray` or nested table type. If your class represents a `REF` type, you can only construct a null `REF`. All non-null `REF`s come from the database.

Once you have initialized your object, you can:

- Pass the object to wrapper methods in other classes.
- Use the object as a host variable in your own `#sql` statements.
- Use the object as a host variable in your own JDBC calls.

- Call the methods that read and write the object's state. These methods operate on the collection object in your program and do not affect data in the database.
  - For a class that represents an object type, you can call the `get` and `set` 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 the collection as a whole. The `getElement()` and `setElement()` methods return or modify individual elements of the collection.
- You cannot modify a REF, because it is an immutable entity, but you can read and write the database object it references, using the `getValue()` and `setValue()` methods.

  The `getValue()` method returns a copy of the database object to which the REF refers. The `setValue()` method updates an object type instance in the database, taking as input an instance of the Java class that represents the object type. Unlike the `get` and `set` accessor methods of a class generated for an object type, the `getValue()` and `setValue()` methods read and write database objects.

A few methods have not been mentioned yet. You use the `getFactory()` method in JDBC code to return a `CustomDatumFactory` that constructs objects of your JPublisher-generated class. You pass this `CustomDatumFactory` to the Oracle `getCustomDatum()` methods in the classes `ArrayDataResultSet`, `OracleCallableStatement`, and `OracleResultSet` classes in the `oracle.jdbc.driver` package. The Oracle JDBC driver uses the `CustomDatumFactory` to create an object of your class.

In addition, classes representing `varray`s 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`.

The following methods that JPublisher generates and Oracle's JDBC drivers employ are not intended for your use:

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

These methods implement the `CustomDatum` and `CustomDatumFactory` interfaces, which the Oracle JDBC drivers require.

The `RationalP` example, described in "Example: Using Classes Generated for Packages" on page 2-65, includes a class that was generated for an object type that does not have wrapper methods.

# JPublisher Limitations

This section describes some of the limitations in the current release of JPublisher.

**Using Multi-byte Character Sets** JPublisher does not currently fully support multi-byte character sets.

**Error Message Availability** Error messages are currently available only in English.

**Not All PL/SQL Types are Supported** Publisher does not currently support the following PL/SQL argument types:

| Unsupported PL/SQL Type Name |
|---|
| BOOLEAN |
| NCHAR |
| NCLOB |
| NVARCHAR2 |
| record types |
| index-by tables |
| strongly-typed REF CURSORs |

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

**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 1-45.

# 2

# JPublisher Examples

This chapter describes the output JPublisher produces when translating object types and PL/SQL packages. This chapter contains these sections:

- Example: JPublisher Translations with Different Mappings contains examples of JPublisher output where only the value of the type mapping parameters is changed.
- Example: JPublisher Type 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.
- Examples: JPublisher Wrapper Methods shows an example of JPublisher output when generating method wrappers for PL/SQL methods, and for object type attributes and methods.
- Example: Using Classes Generated for Object Types describes 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 value of the type mapping parameters. This section uses the following SQL type declaration presented in "Sample JPublisher Translation" on page 1-10:

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

with the original command line:

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

In the example on page 1-10, the types were translated with `-numbertypes=objectjdbc` and `-builtintypes=jdbc`.

In the following two examples, JPublisher translates the types using different type mapping options: first, with `-numbertypes=jdbc`, and second with `-numbertypes=oracle` and `-builtintypes=oracle`.

## JPublisher Translation with the JDBC mapping

The SQL program presented in "Sample JPublisher Translation" on page 1-10 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 `get` and `set` accessor methods use the type `int` instead of `Integer` and the type `float` instead of `Float`.

Following are the contents of the `Employee.sqlj` 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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
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 Employee implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.EMPLOYEE";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  #sql static context _Ctx;
  _Ctx _ctx;

  MutableStruct _struct;

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

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

  static final Employee _EmployeeFactory = new Employee();
  public static CustomDatumFactory getFactory()
  {
    return _EmployeeFactory;
  }

  /* constructors */
  public Employee()
```

```
{
  _struct = new MutableStruct(new Object[5], _sqlType, _factory);
  try
  {
    _ctx = new _Ctx(DefaultContext.getDefaultContext());
  }
  catch (Exception e)
  {
    _ctx = null;
  }
}

public Employee(ConnectionContext c) throws SQLException
{
  _struct = new MutableStruct(new Object[5], _sqlType, _factory);
  _ctx = new _Ctx(c == null ? DefaultContext.getDefaultContext()
                            : c);
}
public Employee(Connection c) throws SQLException
{
  _struct = new MutableStruct(new Object[5], _sqlType, _factory);
  _ctx = new _Ctx(c);
}

/* CustomDatum interface */
public Datum toDatum(OracleConnection c) throws SQLException
{
  _ctx = new _Ctx(c);
  return _struct.toDatum(c, _SQL_NAME);
}

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Employee o = new Employee();
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o._ctx = new _Ctx(((STRUCT) d).getConnection());
  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 1-10 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 `get` and `set` 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.sqlj` 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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
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 Employee implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.EMPLOYEE";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  #sql static context _Ctx;
  _Ctx _ctx;

  MutableStruct _struct;

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

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

  static final Employee _EmployeeFactory = new Employee();
  public static CustomDatumFactory getFactory()
  {
    return _EmployeeFactory;
```

```
}

/* constructors */
public Employee()
{
  _struct = new MutableStruct(new Object[5], _sqlType, _factory);
  try
  {
    _ctx = new _Ctx(DefaultContext.getDefaultContext());
  }
  catch (Exception e)
  {
    _ctx = null;
  }
}

public Employee(ConnectionContext c) throws SQLException
{
  _struct = new MutableStruct(new Object[5], _sqlType, _factory);
  _ctx = new _Ctx(c == null ? DefaultContext.getDefaultContext()
                            : c);
}
public Employee(Connection c) throws SQLException
{
  _struct = new MutableStruct(new Object[5], _sqlType, _factory);
  _ctx = new _Ctx(c);
}

/* CustomDatum interface */
public Datum toDatum(OracleConnection c) throws SQLException
{
  _ctx = new _Ctx(c);
  return _struct.toDatum(c, _SQL_NAME);
}

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Employee o = new Employee();
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o._ctx = new _Ctx(((STRUCT) d).getConnection());
  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 Type Mapping

This section describes an example of JPublisher output for a variety of object attribute types. The example demonstrates the various type 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 REFs 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, invoke JPublisher with the following command line:

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

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 GENERATE 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 `REF` type for each object type. A `REF` type is in the same package as the corresponding object type. `REF` 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
```

## JPublisher Type Mapping Example Output

This section lists the contents of the files JPublisher produces.

### 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.

```
package corp;

import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

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

  MutableStruct _struct;

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

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

  static final Address _AddressFactory = new Address();
  public static CustomDatumFactory getFactory()
  {
    return _AddressFactory;
  }

  /* constructor */
  public Address()
```

```
{
  _struct = new MutableStruct(new Object[4], _sqlType, _factory);
}

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

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Address o = new Address();
  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. Java source files JPublisher generates 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:

- `-package` parameter that you specify on the command line or in the properties file
- `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 `getFactory()`. Other `Address` objects are constructed by the `create()` method. The static `_factory` field is an array of factories for the attributes of `Address`, because none of the attribute types of `Address` require a factory:

`_factory[0] == _factory[1] == null`.

The actual data is stored in the `MutableStruct _struct`.

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

The `get` and `set` 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 and Description 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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class AddressRef implements CustomDatum, CustomDatumFactory
{
  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 CustomDatumFactory getFactory()
  {
    return _AddressRefFactory;
  }

  /* constructor */
  public AddressRef()
  {
  }

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

  /* CustomDatumFactory interface */
  public CustomDatum 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.getFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

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

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

### Listing and Description 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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Alltypes implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.ALLTYPES";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;
```

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

static final Alltypes _AlltypesFactory = new Alltypes();
public static CustomDatumFactory getFactory()
{
  return _AlltypesFactory;
}

/* constructor */
public Alltypes()
{
  _struct = new MutableStruct(new Object[20], _sqlType, _factory);
}

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

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Alltypes o = new Alltypes();
  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 `oracle.sql`. 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.

### Listing and Description 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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class AlltypesRef implements CustomDatum, CustomDatumFactory
{
  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 CustomDatumFactory getFactory()
```

```
  {
    return _AlltypesRefFactory;
  }

  /* constructor */
  public AlltypesRef()
  {
  }

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

  /* CustomDatumFactory interface */
  public CustomDatum 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.getFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

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

### Listing and Description 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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;

public class Ntbl implements CustomDatum, CustomDatumFactory
{
  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 CustomDatumFactory getFactory()
  {
    return _NtblFactory;
  }

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

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

  /* CustomDatum interface */
```

```
public Datum toDatum(OracleConnection c) throws SQLException
{
  return _array.toDatum(c, _SQL_NAME);
}

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Ntbl a = new Ntbl();
  a._array = new MutableArray((ARRAY) d, 4, 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 and Description of AddrArray.java Generated by JPublisher

JPublisher generates declarations of the types `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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;
```

```
public class AddrArray implements CustomDatum, CustomDatumFactory
{
  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 CustomDatumFactory getFactory()
  {
    return _AddrArrayFactory;
  }

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

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

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

  /* CustomDatumFactory interface */
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    AddrArray a = new AddrArray();
    a._array = new MutableArray((ARRAY) d, 2002, Address.getFactory());
    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 a CustomDatum class. The command line for this example is:

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

The option `-usertypes=jdbc` instructs JPublisher to generate classes that implement the SQLData interface. The SQLData interface supports REF and collection classes generically, using the generic classes java.sql.Ref and java.sql.Array rather than 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.CustomDatum` 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.driver.OracleConnection;
import oracle.jdbc.driver.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 we specified `-usertypes=jdbc` in this example, the `Alltypes` class implements the `java.sql.SQLData` interface rather than the `oracle.sql.CustomDatum` 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.driver.OracleConnection;
import oracle.jdbc.driver.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 "Extending JPublisher Classes" on page 1-43 describes.

The following is the code that you have written for the class `MyAddress.java` and stored in the directory `demo/corp`.

```
package corp;

import java.sql.SQLException;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
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 */

  /* _sqlType inherited from JAddress */

  /* _factory inherited from JAddress */

  /* _struct inherited from JAddress */

  static final MyAddress _MyAddressFactory = new MyAddress();
  public static CustomDatumFactory getFactory()
  {
    return _MyAddressFactory;
  }

  /* constructor */
  public MyAddress()
  {
    super();
  }

  /* CustomDatum interface */
  /* toDatum() inherited from JAddress */

  /* CustomDatumFactory interface */
  public CustomDatum create(Datum d, int sqlType) throws SQLException
```

```
  {
     if (d == null) return null;
     MyAddress o = new MyAddress();
     o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
     o._ctx = new _Ctx(((STRUCT) d).getConnection());
     return o;
  }

  /* 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 contents of the `demoin` file is:

```
SQL ADDRESS GENERATE JAddress AS MyAddress
```

JPublisher will generate these files:

```
demo/corp/JAddress.java
demo/corp/MyAddressRef.java
```

Because an `ADDRESS` will be represented in the Java program as a `MyAddress` class, 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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
```

```
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class JAddress implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  MutableStruct _struct;

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

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

  static final JAddress _JAddressFactory = new JAddress();
  public static CustomDatumFactory getFactory()
  {
    return _JAddressFactory;
  }

  /* constructor */
  public JAddress()
  {
    _struct = new MutableStruct(new Object[4], _sqlType, _factory);
  }

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

  /* CustomDatumFactory interface */
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    JAddress o = new JAddress();
    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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class MyAddressRef implements CustomDatum, CustomDatumFactory
{
  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 CustomDatumFactory getFactory()
  {
    return _MyAddressRefFactory;
  }

  /* constructor */
  public MyAddressRef()
  {
  }

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

  /* CustomDatumFactory interface */
  public CustomDatum 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.getFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

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

# Examples: JPublisher Wrapper Methods

This section describes examples of JPublisher output when you translate an object type containing methods and a PL/SQL package containing methods.

**Notes:**

- The wrapper methods that JPublisher generates to invoke stored procedures are in SQLJ only.
- Classes that JPublisher generates containing wrapper methods must be compiled by SQLJ.

## Wrappers Generated for Methods in an Object Type

This section describes an example of JPublisher output given the following definition of a database type containing methods. The example defines a type `Rational` with `numerator` and `denominator` attributes and these 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 numerator and denominator by their greatest common divisor.
- `STATIC FUNCTION gcd`: given two integers, this function returns their greatest common divisor.
- `MEMBER FUNCTION plus`: 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, you invoke JPublisher with the following command line:

```
jpub -user=scott/tiger -sql=Rational -methods=true
```

The `-user` parameter directs JPublisher to log into 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:

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

```
import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
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 CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.RATIONAL";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  #sql static context _Ctx;
  _Ctx _ctx;

  MutableStruct _struct;

  static int[] _sqlType =
  {
    4, 4
  };

  static CustomDatumFactory[] _factory = new CustomDatumFactory[2];

  static final Rational _RationalFactory = new Rational();
  public static CustomDatumFactory getFactory()
  {
```

```
  return _RationalFactory;
}

/* constructors */
public Rational()
{
  _struct = new MutableStruct(new Object[2], _sqlType, _factory);
  try
  {
    _ctx = new _Ctx(DefaultContext.getDefaultContext());
  }
  catch (Exception e)
  {
    _ctx = null;
  }
}

public Rational(ConnectionContext c) throws SQLException
{
  _struct = new MutableStruct(new Object[2], _sqlType, _factory);
  _ctx = new _Ctx(c == null ? DefaultContext.getDefaultContext()
                            : c);
}
public Rational(Connection c) throws SQLException
{
  _struct = new MutableStruct(new Object[2], _sqlType, _factory);
  _ctx = new _Ctx(c);
}

/* CustomDatum interface */
public Datum toDatum(OracleConnection c) throws SQLException
{
  _ctx = new _Ctx(c);
  return _struct.toDatum(c, _SQL_NAME);
}

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Rational o = new Rational();
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o._ctx = new _Ctx(((STRUCT) d).getConnection());
  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 [_ctx] __jPt_result = { VALUES(RATIONAL.GCD(
    :x,
    :y)) };
  return __jPt_result;
}

public Rational normalize ()
throws SQLException
{
  Rational __jPt_temp = this;
  #sql [_ctx] {
    BEGIN
    :INOUT __jPt_temp.NORMALIZE();
    END;
  };
  return __jPt_temp;
}

public Rational plus (
  Rational x)
throws SQLException
{
  Rational __jPt_temp = this;
```

```
    Rational __jPt_result;
    #sql [_ctx] {
      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 [_ctx] {
      BEGIN
      :OUT __jPt_result := :__jPt_temp.TOREAL();
      END;
    };
    return __jPt_result;
  }
}
```

All the methods 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`, and creates a SQLJ connection context `_Ctx`. It creates accessor methods `get/setNumerator()` and `get/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`. It defines a host `OUT` variable that is assigned the value returned by the function. A copy of `this` is an `IN` parameter.

## Wrappers Generated for Methods in Packages

This section describes an example of JPublisher output given the following definition of a PL/SQL package containing methods. The example defines the package `RationalP` with these 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 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, you invoke JPublisher with the following command line:

```
jpub -user=scott/tiger -sql=RationalP -methods=true
```

The `-user` parameter directs JPublisher to log into 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
{

  #sql static context _Ctx;
  _Ctx _ctx;

  /* constructors */
  public RationalP() throws SQLException
  {
    _ctx = new _Ctx(DefaultContext.getDefaultContext());
  }

  public RationalP(ConnectionContext c) throws SQLException
  {
    _ctx = new _Ctx(c);
  }
  public RationalP(Connection c) throws SQLException
  {
    _ctx = new _Ctx(c);
  }

  public Integer gcd (
    Integer x,
    Integer y)
  throws SQLException
  {
```

```
  Integer __jPt_result;
  #sql [_ctx] __jPt_result = { VALUES(RATIONALP.GCD(
    :x,
    :y)) };
  return __jPt_result;
}

public void normalize (
  Integer numerator[],
  Integer denominator[])
throws SQLException
{
  #sql [_ctx] { 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 [_ctx] { 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 [_ctx] __jPt_result = { VALUES(RATIONALP.TOREAL(
    :numerator,
    :denominator)) };
  return __jPt_result;
```

```
  }
}
```

All of the methods that JPublisher generates invoke the corresponding PL/SQL methods executing in the server.

JPublisher creates a SQLJ connection context `_Ctx` and associates 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`, `:d2`. It also defines the `OUT` host variables `:n3` and `:d3`. Because these are `OUT` variables, JPublisher passes 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` 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 and / or compiling the classes, you can use them in a program. For more information on this topic, see "Using Classes JPublisher Generates for Object Types" on page 1-53.

The following steps demonstrate the scenario described above. In this case, you 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, you provide a file, `RationalO.java`, that enhances the functionality of the `JPubRationalO` class by extending it. After compiling the necessary files, you use the classes in a test file to test the performance of the `RationalO.java` class.

The sections following the steps list the contents of the files the steps mention.

1. Create the SQL object type `RationalO`. "Listing of RationalO.sql to Create the Object Type" on page 2-55 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
   ```

   where the properties file `RationalO.props` contains:

   ```
   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 `RatinalO.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 files:

```
JPubRational0.sqlj
Rational0Ref.java
```

See "Listing of JPubRational0.sqlj Generated by JPublisher" on page 2-56 and "Listing of Rational0Ref.java Generated by JPublisher" on page 2-59 for listings of the `JPubRational0.sqlj` and `Rational0Ref.java` files.

3. Write a file `Rational0.java` that enhances the functionality of `JPubRational0.sqlj` by extending it. In `Rational0.java`, everything is inherited from the superclass except the following items. You add code to:
   - declare a factory object, `_JPubRational0`
   - implement a `getFactory()` 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 `TestRational0.java` (described in "Listing of TestRational0.java Written by a User" on page 2-63)

   "Listing of Rational0.java Written by a User" on page 2-61 contains the code for the `Rational0.java` file.

4. Compile / translate the necessary files. Enter:

   ```
   sqlj JPubRational0.sqlj Rational0.java
   ```

   to translate `JPubRational0.sqlj` and compile the `Rational0.java` file.

5. Write a program `TestRational0.java` that uses the `Rational0.java` class. "Listing of TestRational0.java Written by a User" on page 2-63 contains the code for `TestRational0.java`.

6. Write the file `connect.properties`, which `TestRational0.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:oci8:@
   sqlj.driver=oracle.jdbc.driver.OracleDriver
   ```

7. Compile, then run `TestRational0.java`:

   ```
   javac TestRational0.java
   java TestRational0
   ```

The program produces the following output:

```
gcd: 5
real value: 0.5
sum: 100/100
sum: 1/1
```

## Listing of RationalO.sql to Create the 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 for `JPubRationalO.java` that JPublisher generates.

```
import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
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 CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.RATIONALO";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  #sql static context _Ctx;
  _Ctx _ctx;

  MutableStruct _struct;

  static int[] _sqlType =
  {
    4, 4
  };
```

```
static CustomDatumFactory[] _factory = new CustomDatumFactory[2];

static final JPubRational0 _JPubRational0Factory = new JPubRational0();
public static CustomDatumFactory getFactory()
{
  return _JPubRational0Factory;
}

/* constructors */
public JPubRational0()
{
  _struct = new MutableStruct(new Object[2], _sqlType, _factory);
  try
  {
    _ctx = new _Ctx(DefaultContext.getDefaultContext());
  }
  catch (Exception e)
  {
    _ctx = null;
  }
}

public JPubRational0(ConnectionContext c) throws SQLException
{
  _struct = new MutableStruct(new Object[2], _sqlType, _factory);
  _ctx = new _Ctx(c == null ? DefaultContext.getDefaultContext()
                            : c);
}
public JPubRational0(Connection c) throws SQLException
{
  _struct = new MutableStruct(new Object[2], _sqlType, _factory);
  _ctx = new _Ctx(c);
}

/* CustomDatum interface */
public Datum toDatum(OracleConnection c) throws SQLException
{
  _ctx = new _Ctx(c);
  return _struct.toDatum(c, _SQL_NAME);
}

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
```

```
  if (d == null) return null;
  JPubRational0 o = new JPubRational0();
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o._ctx = new _Ctx(((STRUCT) d).getConnection());
  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 [_ctx] __jPt_result = { VALUES(RATIONAL0.GCD(
    :x,
    :y)) };
  return __jPt_result;
}

public Rational0 normalize ()
throws SQLException
{
  Rational0 __jPt_temp = (Rational0) this;
  #sql [_ctx] {
    BEGIN
    :INOUT __jPt_temp.NORMALIZE();
    END;
  };
  return __jPt_temp;
}
```

```
  public RationalO plus (
    RationalO x)
  throws SQLException
  {
    JPubRationalO __jPt_temp = this;
    RationalO __jPt_result;
    #sql [_ctx] {
      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 [_ctx] {
      BEGIN
      :OUT __jPt_result := :__jPt_temp.TOREAL();
      END;
    };
    return __jPt_result;
  }
}
```

## Listing of RationalORef.java Generated by JPublisher

This section lists the code for `RationalORef.java` that JPublisher generates.

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

```
import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
```

```
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class RationalORef implements CustomDatum, CustomDatumFactory
{
  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 CustomDatumFactory getFactory()
  {
    return _RationalORefFactory;
  }

  /* constructor */
  public RationalORef()
  {
  }

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

  /* CustomDatumFactory interface */
  public CustomDatum 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.getFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

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

```
}
```

## Listing of RationalO.java Written by a User

This section lists the code for the user-written file, `RationalO.java`, that extends the class `JPubRationalO.sqlj`. Note that this program:

- declares a factory object, `_JPubRationalO`
- implements a `getFactory()` method
- implements a `create()` method
- implements the constructors by calling the constructors in the superclass
- adds 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 a User" on page 2-63)

```
import java.sql.SQLException;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

public class RationalO extends JPubRationalO
                       implements CustomDatum, CustomDatumFactory
{
  /* _SQL_NAME inherited from JPubRationalO */

  /* _SQL_TYPECODE inherited from JPubRationalO */

  /* _ctx inherited from JPubRationalO */

  /* _sqlType inherited from JPubRationalO */

  /* _factory inherited from JPubRationalO */

  /* _struct inherited from JPubRationalO */

  static final RationalO _RationalOFactory = new RationalO();
  public static CustomDatumFactory getFactory()
```

```
{
  return _RationalOFactory;
}

/* constructors */
public RationalO()
{
  super();
}

public RationalO(ConnectionContext c) throws SQLException
{
  super(c);
}

public RationalO(Connection c) throws SQLException
{
  super(c);
}

/* CustomDatum interface */
/* toDatum() inherited from JPubRationalO */

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  RationalO o = new RationalO();
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o._ctx = new _Ctx(((STRUCT) d).getConnection());
  return o;
}

/* 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 a User

This section lists the contents of a user-written file, `TestRationalO.java`, that tests the performance of the `RationalO.java` class, given initial values for `numerator` and `denominator`. Note that the `TestRationalO.java` file also demonstrates how to:

- connect to the database by calling `Oracle.connect`
- declare a Java object representing a database 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 illustrates an example of how you can use the classes and method wrappers that JPublisher generates for objects and packages respectively. Suppose that 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 "Using SQLJ Classes JPublisher Generates for PL/SQL Packages" on page 1-53.

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, you translate them with SQLJ, then use them in a test file to test the performance of the `Rational` and `RationalP` classes.

The sections following the steps list the contents of the files the steps mention.

1. Create the SQL object type `Rational` and package `RationalP`. "Listing of RationalP.sql to Create the Object Type and Package" on page 2-66 contains the SQL code for the `RationalP.sql` file.

2. Use JPublisher to generate SQLJ classes (`RationalP.sqlj` and `Rational.sqlj` files) for the object and package. Use this command line:

   ```
   jpub -props=RationalP.props
   ```

   where the properties file `RationalP.props` contains:

   ```
   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 and wrapper methods.

   JPublisher produces the files:

   ```
   RationalP.sqlj
   Rational.sqlj
   ```

3. Translate the `RationalP.sqlj` and `Rational.sqlj` files:

   ```
   sqlj RationalP.sqlj Rational.sqlj
   ```

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:oci8:@
   sqlj.driver=oracle.jdbc.driver.OracleDriver
   ```

6. Compile, then run `TestRationalP.java`:

   ```
   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 to Create the Object Type and Package

This section lists the contents of the file `RationalP.sql` which creates 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 a couple of `Rational` objects.

`TestRationalP.java` connects to the database in SQLJ style, using `Oracle.connect()`. In this example, `Oracle.connect()` specifies the file `connect.properties`, which contains these connection properties:

```
sqlj.url=jdbc:oracle:oci8:@
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 oracle.jdbc.driver.OracleConnection;

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 that 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 1-50.

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 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 it 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, or `IN OUT` argument, or function result from the PL/SQL block.
7. In Java, convert each `OUT` argument or `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, then modified by a user, according to the steps above. The wrapper methods 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, or `IN OUT` argument, or function result from `BOOLEAN` to `VARCHAR2` in PL/SQL; return each `OUT` argument, or `IN OUT` argument, or function result from the PL/SQL block; and finally, convert each `OUT` argument, or `IN OUT` argument, or function result:

```
import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
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 CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.BOOLEANS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  #sql static context _Ctx;
  _Ctx _ctx;

  MutableStruct _struct;

  static int[] _sqlType =
  {
    4, 4, 4
  };

  static CustomDatumFactory[] _factory = new CustomDatumFactory[3];

  static final Booleans _BooleansFactory = new Booleans();
  public static CustomDatumFactory getFactory()
  {
    return _BooleansFactory;
  }

  /* constructors */
  public Booleans()
  {
    _struct = new MutableStruct(new Object[3], _sqlType, _factory);
    try
    {
      _ctx = new _Ctx(DefaultContext.getDefaultContext());
    }
    catch (Exception e)
    {
```

```
    _ctx = null;
  }
}

public Booleans(ConnectionContext c) throws SQLException
{
  _struct = new MutableStruct(new Object[3], _sqlType, _factory);
  _ctx = new _Ctx(c == null ? DefaultContext.getDefaultContext()
                            : c);
}
public Booleans(Connection c) throws SQLException
{
  _struct = new MutableStruct(new Object[3], _sqlType, _factory);
  _ctx = new _Ctx(c);
}

/* CustomDatum interface */
public Datum toDatum(OracleConnection c) throws SQLException
{
  _ctx = new _Ctx(c);
  return _struct.toDatum(c, _SQL_NAME);
}

/* CustomDatumFactory interface */
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Booleans o = new Booleans();
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o._ctx = new _Ctx(((STRUCT) d).getConnection());
  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); }


public Boolean f (
  Boolean i1)
throws SQLException
{
  Booleans _temp = this;
  String _i1 = null;
  String _result = null;

  if (i1 != null) _i1 = i1.toString();

  #sql [_ctx] {
    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"));
}

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 [_ctx] {
    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

## M



## N



## O



## P



## R

## S



## T



## U



## V



## W