6

Publishing Java Classes With Call Specifications

When you load a Java class into the database, its methods are not published automatically, because Oracle Database does not know which methods are safe entry points for calls from SQL. To publish the methods, you must write call specifications, which map Java method names, parameter types, and return types to their SQL counterparts. This chapter describes how to publish Java classes with call specifications in the following sections:

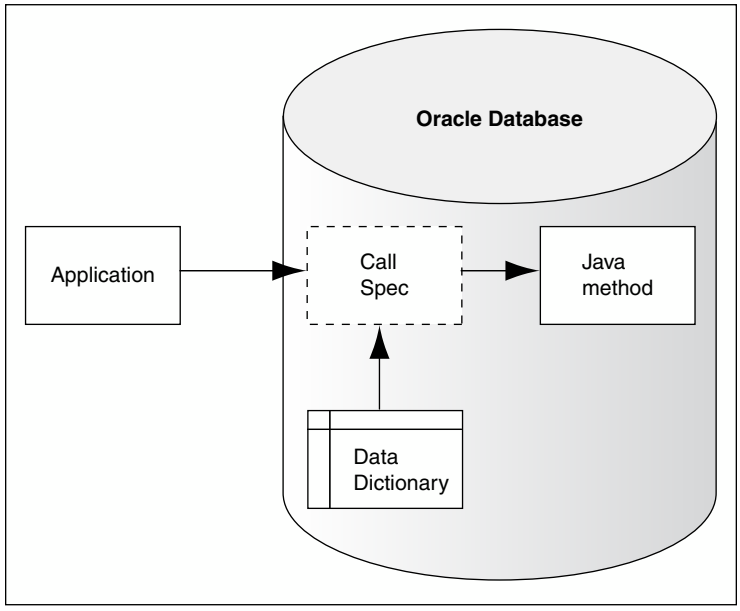
6.1 What Are Call Specifications?

To publish Java methods, you write call specifications. For a given Java method, you declare a function or procedure call specification using the SQL `CREATE FUNCTION` or `CREATE PROCEDURE` statement. Inside a PL/SQL package or SQL object type, you use similar declarations.

You publish Java methods that return a value as functions and `void` Java methods as procedures. The function or procedure body contains the `LANGUAGE JAVA` clause. This clause records information about the Java method including its full name, its parameter types, and its return type. Mismatches are detected only at run time.

The following figure shows applications calling the Java method through its call specification, that is, by referencing the name of the call specification. The run-time system looks up the call specification definition in the Oracle data dictionary and runs the corresponding Java method.

Figure 6-1 Calling a Java Method



As an alternative, you can use the native Java interface to directly call Java methods in the database from a Java client.

Related Topics

- Overview of Using the Client-Side Stub

6.2 Defining Call Specifications

A call specification and the Java method it publishes must reside in the same schema, unless the Java method has a `PUBLIC` synonym. You can declare the call specification as a:

- Standalone PL/SQL function or procedure
- Packaged PL/SQL function or procedure
- Member method of a SQL object type

A call specification exposes the top-level entry point of a Java method to Oracle Database. As a result, you can publish only `public static` methods. However, there is an exception. You can publish instance methods as member methods of a SQL object type.

Packaged call specifications perform as well as top-level call specifications. As a result, to ease maintenance, you may want to place call specifications in a package body. This will help you to modify call specifications without invalidating other schema objects. Also, you can overload the call specifications.

This section covers the following topics:

- About Setting Parameter Modes
- About Mapping Data Types
- Using the Server-Side Internal JDBC Driver

6.2.1 About Setting Parameter Modes

In Java and other object-oriented languages, a method cannot assign values to objects passed as arguments. When calling a method from SQL or PL/SQL, to change the value of an argument, you must declare it as an `OUT` or `IN OUT` parameter in the call specification. The corresponding Java parameter must be an array with only one element.

You can replace the element value with another Java object of the appropriate type, or you can modify the value, if the Java type permits. Either way, the new value propagates back to the caller. For example, you map a call specification `OUT` parameter of the `NUMBER` type to a Java parameter declared as `float[] p`, and then assign a new value to `p[0]`.

> **Note:**
>
> A function that declares `OUT` or `IN OUT` parameters cannot be called from SQL data manipulation language (DML) statements.

6.2.2 About Mapping Data Types

In a call specification, the corresponding SQL and Java parameters and function results must have compatible data types.

Table 6-1 lists the legal data type mappings. Oracle Database converts between the SQL types and Java classes automatically.

Table 6-1 Legal Data Type Mappings

SQL Type	Java Class
`CHAR`, `VARCHAR2`, `LONG`	`java.lang.String`
	`oracle.sql.CHAR`
	`oracle.sql.ROWID`
	`byte[]`

Table 6-1 (Cont.) Legal Data Type Mappings

SQL Type	Java Class
NUMBER	boolean
	char
	byte
	byte[]
	short
	int
	long
	float
	double
	java.lang.Byte
	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	oracle.sql.NUMBER
BINARY_INTEGER	boolean
	char
	byte
	byte[]
	short
	int
	long
BINARY_FLOAT	oracle.sql.BINARY_FLOAT
	byte[]
BINARY_DOUBLE	oracle.sql.BINARY_DOUBLE
	byte[]
DATE	oracle.sql.DATE
	byte[]
RAW	oracle.sql.RAW
	byte[]
BLOB	oracle.sql.BLOB
CLOB	oracle.sql.CLOB
BFILE	oracle.sql.BFILE
ROWID	oracle.sql.ROWID
	byte[]
TIMESTAMP	oracle.sql.TIMESTAMP
	byte[]
TIMESTAMP WITH TIME ZONE	oracle.sql.TIMESTAMPTZ

Table 6-1 (Cont.) Legal Data Type Mappings

SQL Type	Java Class
TIMESTAMP WITH LOCAL TIME ZONE	oracle.sql.TIMESTAMPLTZ
ref cursor	java.sql.ResultSet
user defined named types, ADTs	oracle.sql.STRUCT
opaque named types	oracle.sql.OPAQUE
nested tables and VARRAY named types	oracle.sql.ARRAY
references to named types	oracle.sql.REF

You also must consider the following:

- The last four SQL types are collectively referred to as named types.
- All SQL types except BLOB, CLOB, BFILE, REF CURSOR, and the named types can be mapped to the Java type byte[], which is a Java byte array. In this case, the argument conversion means copying the raw binary representation of the SQL value to or from the Java byte array.
- Java classes that implement the ORAData interface and related methods, or Java classes that are subclasses of the oracle.sql classes appearing in the table, can be mapped from SQL types other than BINARY_INTEGER and REF CURSOR.
- The UROWID type and the NUMBER subtypes, such as INTEGER and REAL, are not supported.
- A value larger than 32 KB cannot be retrieved from a LONG or LONG RAW column into a Java stored procedure.

6.2.3 Using the Server-Side Internal JDBC Driver

Java Database Connectivity (JDBC) enables you establish a connection to the database using the DriverManager class, which manages a set of JDBC drivers. You can use the getConnection() method after loading the JDBC drivers. When the getConnection() method finds the right driver, it returns a Connection object that represents a database session. All SQL statements are run within the context of that session.

However, the server-side internal JDBC driver runs within a default session and a default transaction context. As a result, you are already connected to the database, and all your SQL operations are part of the default transaction. You need not register the driver because it comes preregistered. To get a Connection object, run the following line of code:

```
Connection conn = DriverManager.getConnection("jdbc:default:connection:");
```

Use the Statement class for SQL statements that do not take IN parameters and are run only once. When called on a Connection object, the createStatement() method returns a new Statement object, as follows:

```
String sql = "DROP " + object_type + " " + object_name;
Statement stmt = conn.createStatement();
stmt.executeUpdate(sql);
```

Use the `PreparedStatement` class for SQL statements that take `IN` parameters or are run more than once. The SQL statement, which can contain one or more parameter placeholders, is precompiled. A question mark (?) serves as a placeholder. When called on a `Connection` object, the `prepareStatement()` method returns a new `PreparedStatement` object, which contains the precompiled SQL statement. For example:

```
String sql = "DELETE FROM dept WHERE deptno = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setInt(1, deptID);
pstmt.executeUpdate();
```

A `ResultSet` object contains SQL query results, that is, the rows that meet the search condition. You can use the `next()` method to move to the next row, which then becomes the current row. You can use the `get`*`XXX`*`()` methods to retrieve column values from the current row. For example:

```
String sql = "SELECT COUNT(*) FROM " + tabName;
int rows = 0;
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(sql);
while (rset.next())
{
  rows = rset.getInt(1);
}
```

A `CallableStatement` object lets you call stored procedures. It contains the call text, which can include a return parameter and any number of `IN`, `OUT`, and `IN OUT` parameters. The call is written using an escape clause, which is delimited by braces (`{}`). As the following examples show, the escape syntax has three forms:

```
// parameterless stored procedure
CallableStatement cstmt = conn.prepareCall("{CALL proc}");

// stored procedure
CallableStatement cstmt = conn.prepareCall("{CALL proc(?,?)}");

// stored function
CallableStatement cstmt = conn.prepareCall("{? = CALL func(?,?)}");
```

Important Points

When developing JDBC applications that access stored procedures, you must consider the following:

- Each Oracle JVM session has a single implicit native connection to the Database session in which it exists. This connection is conceptual and is not a Java object. It is an inherent aspect of the session and cannot be opened or closed from within the JVM.
- The server-side internal JDBC driver runs within a default transaction context. You are already connected to the database, and all your SQL operations are part of the default transaction. Note that this transaction is a local transaction and not part of a global transaction, such as that implemented by Java Transaction API (JTA) or Java Transaction Service (JTS).
- Statements and result sets persist across calls and their finalizers do not release database cursors. To avoid running out of cursors, close all statements and result sets after you have finished using them. Alternatively, you can ask your DBA to raise the limit set by the initialization parameter, `OPEN_CURSORS`.
- The server-side internal JDBC driver does not support auto-commits. As a result, your application must explicitly commit or roll back database changes.

- You cannot connect to a remote database using the server-side internal JDBC driver. You can connect only to the server running your Java program. For server-to-server connections, use the server-side JDBC Thin driver. For client/server connections, use the client-side JDBC Thin or JDBC Oracle Call Interface (OCI) driver.
- Typically, you should not close the default connection instance because it is a single instance that can be stored in multiple places, and if you close the instance, each would become unusable. If it is closed, a later call to the `OracleDriver.defaultConnection` method gets a new, open instance. The `OracleDataSource.getConnection` method returns a new object every time you call it, but, it does not create a new database connection every time. They all utilize the same implicit native connection and share the same session state, in particular, the local transaction.

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

6.3 Writing Top-Level Call Specifications

This section describes how to define top-level call specifications in SQL*Plus.

In SQL*Plus, you can define top-level call specifications interactively, using the following syntax:

```
CREATE [OR REPLACE]
{ PROCEDURE procedure_name [(param[, param]...)]
| FUNCTION function_name [(param[, param]...)] RETURN sql_type}
[AUTHID {DEFINER | CURRENT_USER}]
[PARALLEL_ENABLE]
[DETERMINISTIC]
{IS | AS} LANGUAGE JAVA
NAME 'method_fullname (java_type_fullname[, java_type_fullname]...)
[return java_type_fullname]';
```

where, `param` is represented by the following syntax:

```
parameter_name [IN | OUT | IN OUT] sql_type
```

The `AUTHID` clause determines the following:

- Whether a stored procedure runs with the privileges of its definer (`AUTHID DEFINER`) or invoker (`AUTHID CURRENT_USER`)
- Whether its unqualified references to schema objects are resolved in the schema of the definer or invoker

If you do not specify the `AUTHID`, then the default behavior is `DEFINER`, that is, the stored procedure runs with the privileges of its definer. You can override the default behavior by specifying the `AUTHID` as `CURRENT_USER`. However, you cannot override the `loadjava` option `-definer` by specifying `CURRENT_USER`.

The `PARALLEL_ENABLE` option declares that a stored function can be used safely in the worker sessions of parallel DML evaluations. The state of a main session is never shared with worker sessions. Each worker session has its own state, which is initialized when the session begins. The function result should not depend on the state of session variables. Otherwise, results might vary across sessions.

The `DETERMINISTIC` option helps the optimizer avoid redundant function calls. If a stored function was called previously with the same arguments, then the optimizer can decide to use the previous result. The function result should not depend on the state of session variables or schema objects. Otherwise, results can vary across calls. Only `DETERMINISTIC` functions can be called from a function-based index or a materialized view that has query-rewrite enabled.

The string in the `NAME` clause uniquely identifies the Java method. The fully-qualified Java names and the call specification parameters, which are mapped by position, must correspond. However, this rule does not apply to the `main()` method. If the Java method does not take any arguments, then write an empty parameter list for it, but not for the function or procedure.

The `method_fullname` portion of the `NAME` clause specifies the fully modularized Java database object name. The Java class database object names, which reside in a module, are of the following format:

```
<module_name>///<class_name>
```

The Java class database object names, which do not reside in a module, are of the following format:

```
<class_name>
```

The `java_type_fullnames`, which are used in return values and method signatures, do not include the `module_name` as a prefix, even if the Java type class names are module-resident.

As an exception to the preceding `method_fullname` rule, if the class specified in the `method_fullname` is a member of a module that is built into the system, then the module name prefixing of `method_fullname` is optional. For example, the following call specification:

```
create or replace function valueof(n number) return varchar2 as language java
name
'java.base///java.lang.String.valueOf(long) return java.lang.String';
```

Can be written in an equivalent way as the following:

```
create or replace function valueof(n number) return varchar2 as language java
name
'java.lang.String.valueOf(long) return java.lang.String';
```

Write fully-qualified Java names using the dot notation. The following example shows that the fully-qualified names can be broken across lines at dot boundaries:

```
artificialIntelligence.neuralNetworks.patternClassification.
RadarSignatureClassifier.computeRange()
```

6.3.1 Examples

This section provides the following examples:

- Example 6-1
- Example 6-2
- Example 6-3
- Example 6-4

Example 6-1 Publishing a Simple JDBC Stored Procedure

Assume that the executable for the following Java class has been loaded into the database:

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.*;

public class GenericDrop
{
  public static void dropIt(String object_type, String object_name)
                                                          throws SQLException
  {
    // Connect to Oracle using JDBC driver
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    // Build SQL statement
    String sql = "DROP " + object_type + " " + object_name;
    try
    {
      Statement stmt = conn.createStatement();
      stmt.executeUpdate(sql);
      stmt.close();
    }
    catch (SQLException e)
    {
      System.err.println(e.getMessage());
    }
  }
}
```

The `GenericDrop` class has one method, `dropIt()`, which drops any kind of schema object. For example, if you pass the `table` and `employees` arguments to `dropIt()`, then the method drops the database table `employees` from your schema.

The call specification for the `dropIt()` method is as follows:

```
CREATE OR REPLACE PROCEDURE drop_it (obj_type VARCHAR2, obj_name VARCHAR2)
AS LANGUAGE JAVA
NAME 'GenericDrop.dropIt(java.lang.String, java.lang.String)';
```

Note that you must fully qualify the reference to `String`. The `java.lang` package is automatically available to Java programs, but must be named explicitly in the call specifications.

Example 6-2 Publishing the main() Method

As a rule, Java names and call specification parameters must correspond. However, that rule does not apply to the `main()` method. Its `String[]` parameter can be mapped to multiple `CHAR` or `VARCHAR2` call specification parameters. Consider the `main()` method in the following class, which displays its arguments:

```
public class EchoInput
{
  public static void main (String[] args)
  {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

To publish `main()`, write the following call specification:

```
CREATE OR REPLACE PROCEDURE echo_input(s1 VARCHAR2, s2 VARCHAR2, s3 VARCHAR2)
AS LANGUAGE JAVA
NAME 'EchoInput.main(java.lang.String[])';
```

You cannot impose constraints, such as precision, size, and `NOT NULL`, on the call specification parameters. As a result, you cannot specify a maximum size for the `VARCHAR2` parameters. However, you must do so for `VARCHAR2` variables, as in:

```
DECLARE last_name VARCHAR2(20); -- size constraint required
```

Example 6-3 Publishing a Method That Returns an Integer Value

In the following example, the `rowCount()` method, which returns the number of rows in a given database table, is published:

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.*;

public class RowCounter
{
  public static int rowCount (String tabName) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    String sql = "SELECT COUNT(*) FROM " + tabName;
    int rows = 0;
    try
    {
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery(sql);
      while (rset.next())
      {
        rows = rset.getInt(1);
      }
      rset.close();
      stmt.close();
    }
    catch (SQLException e)
    {
      System.err.println(e.getMessage());
    }
    return rows;
  }
}
```

`NUMBER` subtypes, such as `INTEGER`, `REAL`, and `POSITIVE`, are not allowed in a call specification. As a result, in the following call specification, the return type is `NUMBER` and not `INTEGER`:

```
CREATE FUNCTION row_count (tab_name VARCHAR2) RETURN NUMBER
AS LANGUAGE JAVA
NAME 'RowCounter.rowCount(java.lang.String) return int';
```

Example 6-4 Publishing a Method That Switches the Values of Its Arguments

Consider the `swap()` method in the following `Swapper` class, which switches the values of its arguments:

```
public class Swapper
{
  public static void swap (int[] x, int[] y)
  {
    int hold = x[0];
```

```
    x[0] = y[0];
    y[0] = hold;
  }
}
```

The call specification publishes the `swap()` method as a call specification, `swap()`. The call specification declares `IN OUT` formal parameters, because values must be passed in and out. All call specification `OUT` and `IN OUT` parameters must map to Java array parameters.

```
CREATE PROCEDURE swap (x IN OUT NUMBER, y IN OUT NUMBER)
AS LANGUAGE JAVA
NAME 'Swapper.swap(int[], int[])';
```

> **Note:**
>
> A Java method and its call specification can have the same name.

6.4 Writing Packaged Call Specifications

A PL/SQL package is a schema object that groups logically related types, items, and subprograms. Usually, packages have two parts, a specification and a body. The specification is the interface to your applications and declares the types, constants, variables, exceptions, cursors, and subprograms available for use. The body defines the cursors and subprograms.

In SQL*Plus, you can define PL/SQL packages interactively, using the following syntax:

```
CREATE [OR REPLACE] PACKAGE package_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
  [type_definition [type_definition] ...]
  [cursor_spec [cursor_spec] ...]
  [item_declaration [item_declaration] ...]
  [{subprogram_spec | call_spec} [{subprogram_spec | call_spec}]...]
END [package_name];

[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
  [type_definition [type_definition] ...]
  [cursor_body [cursor_body] ...]
  [item_declaration [item_declaration] ...]
  [{subprogram_spec | call_spec} [{subprogram_spec | call_spec}]...]
[BEGIN
  sequence_of_statements]
END [package_name];]
```

The specification holds public declarations, which are visible to your application. The body contains implementation details and private declarations, which are hidden from your application. Following the declarative part of the package is the body, which is the optional initialization part. It holds statements that initialize package variables. It is run only once, the first time you reference the package.

A call specification declared in a package specification cannot have the same signature, that is, the name and parameter list, as a subprogram in the package body. If you declare all the subprograms in a package specification as call specifications, then the package body is not required, unless you want to define a cursor or use the initialization part.

The `AUTHID` clause determines whether all the packaged subprograms run with the privileges of their definer (`AUTHID DEFINER`), which is the default, or invoker (`AUTHID CURRENT_USER`). It

also determines whether unqualified references to schema objects are resolved in the schema of the definer or invoker.

Example 6-5 provides an example of packaged call specification.

Example 6-5 Packaged Call Specification

Consider a Java class, `DeptManager`, which consists of methods for adding a new department, dropping a department, and changing the location of a department. Note that the `addDept()` method uses a database sequence to get the next department number. The three methods are logically related, and therefore, you may want to group their call specifications in a PL/SQL package.

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.*;

public class DeptManager
{
  public static void addDept (String deptName, String deptLoc) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    String sql = "SELECT deptnos.NEXTVAL FROM dual";
    String sql2 = "INSERT INTO dept VALUES (?, ?, ?)";
    int deptID = 0;
    try
    {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      ResultSet rset = pstmt.executeQuery();
      while (rset.next())
      {
        deptID = rset.getInt(1);
      }
      pstmt = conn.prepareStatement(sql2);
      pstmt.setInt(1, deptID);
      pstmt.setString(2, deptName);
      pstmt.setString(3, deptLoc);
      pstmt.executeUpdate();
      rset.close();
      pstmt.close();
    }
    catch (SQLException e)
    {
      System.err.println(e.getMessage());
    }
  }

  public static void dropDept (int deptID) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    String sql = "DELETE FROM dept WHERE deptno = ?";
    try
    {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, deptID);
      pstmt.executeUpdate();
      pstmt.close();
    }
    catch (SQLException e)
    {
      System.err.println(e.getMessage());
    }
```

```
  }

  public static void changeLoc (int deptID, String newLoc) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    String sql = "UPDATE dept SET loc = ? WHERE deptno = ?";
    try
    {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, newLoc);
      pstmt.setInt(2, deptID);
      pstmt.executeUpdate();
      pstmt.close();
    }
    catch (SQLException e)
    {
      System.err.println(e.getMessage());
    }
  }
}
```

Suppose you want to package the methods `addDept()`, `dropDept()`, and `changeLoc()`. First, you must create the package specification, as follows:

```
CREATE OR REPLACE PACKAGE dept_mgmt AS
PROCEDURE add_dept (dept_name VARCHAR2, dept_loc VARCHAR2);
PROCEDURE drop_dept (dept_id NUMBER);
PROCEDURE change_loc (dept_id NUMBER, new_loc VARCHAR2);
END dept_mgmt;
```

Then, you must create the package body by writing the call specifications for the Java methods, as follows:

```
CREATE OR REPLACE PACKAGE BODY dept_mgmt AS
PROCEDURE add_dept (dept_name VARCHAR2, dept_loc VARCHAR2)
AS LANGUAGE JAVA
NAME 'DeptManager.addDept(java.lang.String, java.lang.String)';

PROCEDURE drop_dept (dept_id NUMBER)
AS LANGUAGE JAVA
NAME 'DeptManager.dropDept(int)';

PROCEDURE change_loc (dept_id NUMBER, new_loc VARCHAR2)
AS LANGUAGE JAVA
NAME 'DeptManager.changeLoc(int, java.lang.String)';
END dept_mgmt;
```

To reference the stored procedures in the `dept_mgmt` package, use the dot notation, as follows:

```
CALL dept_mgmt.add_dept('PUBLICITY', 'DALLAS');
```

6.5 Writing Object Type Call Specifications

In SQL, object-oriented programming is based on object types, which are user-defined composite data types that encapsulate a data structure along with the functions and procedures required to manipulate the data. The variables that form the data structure are known as attributes. The functions and procedures that characterize the behavior of the object type are known as methods, which can be written in Java.

As with a package, an object type has two parts: a specification and a body. The specification is the interface to your applications and declares a data structure, which is a set of attributes,

along with the operations or methods required to manipulate the data. The body implements the specification by defining PL/SQL subprogram bodies or call specifications.

If the specification declares only attributes or call specifications, then the body is not required. If you implement all your methods in Java, then you can place their call specifications in the specification part of the object type and omit the body part.

In SQL*Plus, you can define SQL object types interactively, using the following syntax:

```
CREATE [OR REPLACE] TYPE type_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS} OBJECT (
  attribute_name data_type[, attribute_name data_type]...
  [{MAP | ORDER} MEMBER {function_spec | call_spec},]
  [{MEMBER | STATIC} {subprogram_spec | call_spec}
  [, {MEMBER | STATIC} {subprogram_spec | call_spec}]...]
);

[CREATE [OR REPLACE] TYPE BODY type_name {IS | AS}
  { {MAP | ORDER} MEMBER function_body;
   | {MEMBER | STATIC} {subprogram_body | call_spec};}
  [{MEMBER | STATIC} {subprogram_body | call_spec};]...
END;]
```

The `AUTHID` clause determines whether all member methods of the type run with the privileges of their definer (`AUTHID DEFINER`), which is the default, or invoker (`AUTHID CURRENT_USER`). It also determines whether unqualified references to schema objects are resolved in the schema of the definer or invoker.

This section covers the following topics:

- About Attributes
- Declaring Methods

6.5.1 About Attributes

In an object type specification, all attributes must be declared before any methods are. In addition, you must declare at least one attribute. The maximum number of attributes that can be declared is 1000. Methods are optional.

As with a Java variable, you declare an attribute with a name and data type. The name must be unique within the object type, but can be reused in other object types. The data type can be any SQL type, except `LONG`, `LONG RAW`, `NCHAR`, `NVARCHAR2`, `NCLOB`, `ROWID`, and `UROWID`.

You cannot initialize an attribute in its declaration using the assignment operator or `DEFAULT` clause. Furthermore, you cannot impose the `NOT NULL` constraint on an attribute. However, objects can be stored in database tables on which you can impose constraints.

6.5.2 Declaring Methods

After declaring attributes, you can declare methods. `MEMBER` methods accept a built-in parameter known as `SELF`, which is an instance of the object type. Whether declared implicitly or explicitly, it is always the first parameter passed to a `MEMBER` method. In the method body, `SELF` denotes the object whose method was called. `MEMBER` methods are called on instances, as follows:

```
instance_expression.method()
```

`STATIC` methods, which cannot accept or reference `SELF`, are invoked on the object type and not its instances, as follows:

```
object_type_name.method()
```

If you want to call a Java method that is not `static`, then you must specify the keyword `MEMBER` in its call specification. Similarly, if you want to call a `static` Java method, then you must specify the keyword `STATIC` in its call specification.

This section contains the following topics:

- Map and Order Methods
- Constructor Methods
- Examples

6.5.2.1 Map and Order Methods

The values of a SQL scalar data type, such as `CHAR`, have a predefined order and, therefore, can be compared with other values. However, instances of an object type have no predefined order. To put them in order, SQL calls a user-defined `map` method.

SQL uses the ordering to evaluate boolean expressions, such as `x > y`, and to make comparisons implied by the `DISTINCT`, `GROUP BY`, and `ORDER BY` clauses. A map method returns the relative position of an object in the ordering of all such objects. An object type can contain only one map method, which must be a function without any parameters and with one of the following return types: `DATE`, `NUMBER`, or `VARCHAR2`.

Alternatively, you can supply SQL with an `order` method, which compares two objects. An `order` method takes only two parameters: the built-in parameter, `SELF`, and another object of the same type. If `o1` and `o2` are objects, then a comparison, such as `o1 > o2`, calls the `order` method automatically. The method returns a negative number, zero, or a positive number signifying that `SELF` is less than, equal to, or greater than the other parameter, respectively. An object type can contain only one `order` method, which must be a function that returns a numeric result.

You can declare a `map` method or an `order` method, but not both. If you declare either of these methods, then you can compare objects in SQL and PL/SQL. However, if you do not declare both methods, then you can compare objects only in SQL and solely for equality or inequality.

> **Note:**
>
> Two objects of the same type are equal if the values of their corresponding attributes are equal.

6.5.2.2 Constructor Methods

Every object type has a constructor, which is a system-defined function with the same name as the object type. The constructor initializes and returns an instance of that object type.

Oracle Database generates a default constructor for every object type. The formal parameters of the constructor match the attributes of the object type. That is, the parameters and attributes are declared in the same order and have the same names and data types. SQL never calls a

constructor implicitly. As a result, you must call it explicitly. Constructor calls are allowed wherever function calls are allowed.

> **Note:**
>
> To invoke a Java constructor from SQL, you must wrap calls to it in a `static` method and declare the corresponding call specification as a `STATIC` member of the object type.

6.5.2.3 Examples

In this section, each example builds on the previous one. To begin, you create two SQL object types to represent departments and employees. First, you write the specification for the object type `Department`. The body is not required, because the specification declares only attributes. The specification is as follows:

```
CREATE TYPE Department AS OBJECT (
deptno NUMBER(2),
dname VARCHAR2(14),
loc VARCHAR2(13)
);
```

Then, you create the object type `Employee`. The `deptno` attribute stores a handle, called a `REF`, to objects of the type `Department`. A `REF` indicates the location of an object in an object table, which is a database table that stores instances of an object type. The `REF` does not point to a specific instance copy in memory. To declare a `REF`, you specify the data type `REF` and the object type that `REF` targets. The `Employee` type is created as follows:

```
CREATE TYPE Employee AS OBJECT (
empno NUMBER(4),
ename VARCHAR2(10),
job VARCHAR2(9),
mgr NUMBER(4),
hiredate DATE,
sal NUMBER(7,2),
comm NUMBER(7,2),
deptno REF Department
);
```

Next, you create the SQL object tables to hold objects of type `Department` and `Employee`. Create the `depts` object table, which will hold objects of the `Department` type. Populate the object table by selecting data from the `dept` relational table and passing it to a constructor, which is a system-defined function with the same name as the object type. Use the constructor to initialize and return an instance of that object type. The `depts` table is created as follows:

```
CREATE TABLE depts OF Department AS
SELECT Department(deptno, dname, loc) FROM dept;
```

Create the `emps` object table, which will hold objects of type `Employee`. The last column in the `emps` object table, which corresponds to the last attribute of the `Employee` object type, holds references to objects of type `Department`. To fetch the references into this column, use the operator `REF`, which takes a table alias associated with a row in an object table as its argument. The `emps` table is created as follows:

```
CREATE TABLE emps OF Employee AS
SELECT Employee(e.employee_id, e.first_name, e.job_id, e.manager_id, e.hire_date,
```

```
e.salary, e.commission_pct,
(SELECT REF(d) FROM departments d WHERE d.department_id = e.department_id))
FROM employees e;
```

Selecting a `REF` returns a handle to an object. It does not materialize the object itself. To do that, you can use methods in the `oracle.sql.REF` class, which supports Oracle object references. This class, which is a subclass of `oracle.sql.Datum`, extends the standard JDBC interface, `oracle.jdbc2.Ref`.

Using Class oracle.sql.STRUCT

To continue, you write a Java stored procedure. The `Paymaster` class has one method, which computes an employee's wages. The `getAttributes()` method defined in the `oracle.sql.STRUCT` class uses the default JDBC mappings for the attribute types. For example, `NUMBER` maps to `BigDecimal`. The `Paymaster` class is created as follows:

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;
import oracle.oracore.*;
import oracle.jdbc2.*;
import java.math.*;

public class Paymaster
{
  public static BigDecimal wages(STRUCT e) throws java.sql.SQLException
  {
    // Get the attributes of the Employee object.
    Object[] attribs = e.getAttributes();
    // Must use numeric indexes into the array of attributes.
    BigDecimal sal = (BigDecimal)(attribs[5]); // [5] = sal
    BigDecimal comm = (BigDecimal)(attribs[6]); // [6] = comm
    BigDecimal pay = sal;
    if (comm != null)
      pay = pay.add(comm);
    return pay;
  }
}
```

Because the `wages()` method returns a value, you write a function call specification for it, as follows:

```
CREATE OR REPLACE FUNCTION wages (e Employee) RETURN NUMBER AS
LANGUAGE JAVA
NAME 'Paymaster.wages(oracle.sql.STRUCT) return BigDecimal';
```

This is a top-level call specification, because it is not defined inside a package or object type.

Implementing the SQLData Interface

To make access to object attributes more natural, create a Java class that implements the `SQLData` interface. To do so, you must provide the `readSQL()` and `writeSQL()` methods as defined by the `SQLData` interface. The JDBC driver calls the `readSQL()` method to read a stream of database values and populate an instance of your Java class. In the following example, you revise `Paymaster` by adding a second method, `raiseSal()`:

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;
```

```
import oracle.oracore.*;
import oracle.jdbc2.*;
import java.math.*;

public class Paymaster implements SQLData
{
  // Implement the attributes and operations for this type.
  private BigDecimal empno;
  private String ename;
  private String job;
  private BigDecimal mgr;
  private Date hiredate;
  private BigDecimal sal;
  private BigDecimal comm;
  private Ref dept;

  public static BigDecimal wages(Paymaster e)
  {
    BigDecimal pay = e.sal;
    if (e.comm != null)
      pay = pay.add(e.comm);
    return pay;
  }

  public static void raiseSal(Paymaster[] e, BigDecimal amount)
  {
    e[0].sal = // IN OUT passes [0]
    e[0].sal.add(amount); // increase salary by given amount
  }

  // Implement SQLData interface.

  private String sql_type;

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

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    empno = stream.readBigDecimal();
    ename = stream.readString();
    job = stream.readString();
    mgr = stream.readBigDecimal();
    hiredate = stream.readDate();
    sal = stream.readBigDecimal();
    comm = stream.readBigDecimal();
    dept = stream.readRef();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeBigDecimal(empno);
    stream.writeString(ename);
    stream.writeString(job);
    stream.writeBigDecimal(mgr);
    stream.writeDate(hiredate);
    stream.writeBigDecimal(sal);
    stream.writeBigDecimal(comm);
    stream.writeRef(dept);
```

```
  }
}
```

You must revise the call specification for `wages()`, as follows, because its parameter has changed from `oracle.sql.STRUCT` to `Paymaster`:

```
CREATE OR REPLACE FUNCTION wages (e Employee) RETURN NUMBER AS
LANGUAGE JAVA
NAME 'Paymaster.wages(Paymaster) return BigDecimal';
```

Because the new method, `raiseSal()`, is `void`, write a procedure call specification for it, as follows:

```
CREATE OR REPLACE PROCEDURE raise_sal (e IN OUT Employee, r NUMBER)
AS LANGUAGE JAVA
NAME 'Paymaster.raiseSal(Paymaster[], java.math.BigDecimal)';
```

Again, this is a top-level call specification.

Implementing Object Type Methods

Assume you decide to drop the top-level call specifications `wages` and `raise_sal` and redeclare them as methods of the object type `Employee`. In an object type specification, all methods must be declared after the attributes. The body of the object type is not required, because the specification declares only attributes and call specifications. The `Employee` object type can be re-created as follows:

```
CREATE TYPE Employee AS OBJECT (
empno NUMBER(4),
ename VARCHAR2(10),
job VARCHAR2(9),
mgr NUMBER(4),
hiredate DATE,
sal NUMBER(7,2),
comm NUMBER(7,2),
deptno REF Department
MEMBER FUNCTION wages RETURN NUMBER
AS LANGUAGE JAVA
NAME 'Paymaster.wages() return java.math.BigDecimal',
MEMBER PROCEDURE raise_sal (r NUMBER)
AS LANGUAGE JAVA
NAME 'Paymaster.raiseSal(java.math.BigDecimal)'
);
```

Then, you revise `Paymaster` accordingly. You need not pass an array to `raiseSal()`, because the SQL parameter `SELF` corresponds directly to the Java parameter `this`, even when `SELF` is declared as `IN OUT`, which is the default for procedures.

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;
import oracle.oracore.*;
import oracle.jdbc2.*;
import java.math.*;

public class Paymaster implements SQLData
{
  // Implement the attributes and operations for this type.
  private BigDecimal empno;
```

```
  private String ename;
  private String job;
  private BigDecimal mgr;
  private Date hiredate;
  private BigDecimal sal;
  private BigDecimal comm;
  private Ref dept;

  public BigDecimal wages()
  {
    BigDecimal pay = sal;
    if (comm != null)
      pay = pay.add(comm);
    return pay;
  }

  public void raiseSal(BigDecimal amount)
  {
    // For SELF/this, even when IN OUT, no array is needed.
    sal = sal.add(amount);
  }

  // Implement SQLData interface.

  String sql_type;

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

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    empno = stream.readBigDecimal();
    ename = stream.readString();
    job = stream.readString();
    mgr = stream.readBigDecimal();
    hiredate = stream.readDate();
    sal = stream.readBigDecimal();
    comm = stream.readBigDecimal();
    dept = stream.readRef();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeBigDecimal(empno);
    stream.writeString(ename);
    stream.writeString(job);
    stream.writeBigDecimal(mgr);
    stream.writeDate(hiredate);
    stream.writeBigDecimal(sal);
    stream.writeBigDecimal(comm);
    stream.writeRef(dept);
  }
}
```