CHAPTER 27

Calling Java from PL/SQL

The Java language, originally designed and promoted by Sun Microsystems and now widely promoted by nearly everyone other than Microsoft, offers an extremely diverse set of programming features, many of which are not available natively in PL/SQL. This chapter introduces the topic of creating and using Java Stored Procedures in Oracle, and shows how you can create and use JSP functionality from PL/SQL.

Oracle and Java

Starting with Oracle8*i* Database, the Oracle Database Server has included a Java Virtual Machine that allows Java programs to run efficiently in the server memory space. Many of the core Java class libraries are bundled with Oracle as well, resulting not only in a formidable weapon in the programmer's arsenal, but also a formidable topic for a PL/SQL book! That's why the objectives for this chapter are limited to the following:

- Providing the information you need to load Java classes into the Oracle database, manage those new database objects, and publish them for use inside PL/SQL.
- Offering a basic tutorial on building Java classes that will provide enough guidance to let you construct simple classes to access underlying Java functionality.

In preview, here is the usual way you will create and expose Java stored procedures:

1. Write the Java source code. You can use any convenient text editor or IDE such as Oracle's JDeveloper.
2. Compile your Java into classes and, optionally, bundle them into *.jar* files. Again, you can use an IDE or Sun's command-line *javac* compiler. (Strictly speaking, this step is optional because you can load the source into Oracle and use the built-in Java compiler.)
3. Load the Java classes into Oracle using the *loadjava* command-line utility or the CREATE JAVA statement.
4. Publish the Java class methods by writing PL/SQL "wrappers" to invoke the Java code.

5. Grant privileges as required on the PL/SQL wrapper.
6. Call the PL/SQL programs from any one of a number of environments, as illustrated in Figure 27-1.

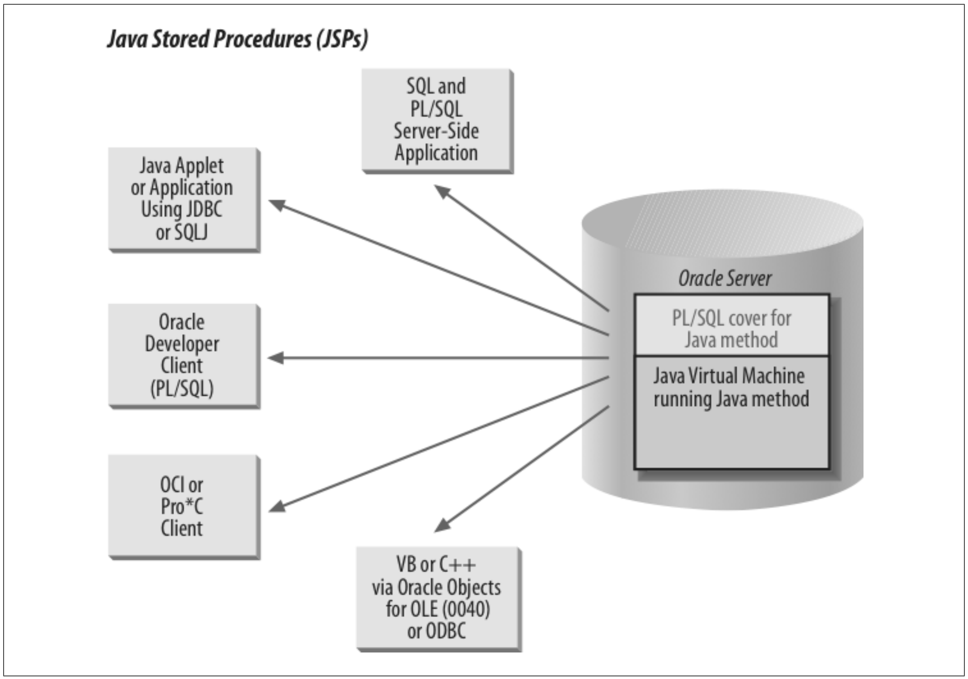

Java Stored Procedures (JSPs)
SQL and PL/SQL Server-Side Application
Java Applet or Application Using JDBC or SQLJ
Oracle Developer Client (PL/SQL)
OCI or Pro*C Client
VB or C++ via Oracle Objects for OLE (0040) or ODBC
Oracle Server
PL/SQL cover for Java method
Java Virtual Machine running Java method

Figure 27-1. Accessing JSPs from within the Oracle database

Oracle offers a variety of components and commands to work with Java. Table 27-1 shows just a few of them.

Table 27-1. Oracle components and commands for Java

Component	Description
"aurora" JVM	The Java Virtual Machine (JVM) that Oracle implemented in its database server
loadjava	An operating system command-line utility that loads your Java code elements (classes, *.jar* files, etc.) into the Oracle database
dropjava	An operating system command-line utility that drops your Java code elements (classes, *.jar* files, etc.) from the Oracle database
CREATE JAVA	DDL statements that perform some of the same tasks as *loadjava* and *dropjava*
DROP JAVA	
ALTER JAVA	
DBMS_JAVA	A built-in package that offers a number of utilities to set options and other aspects of the JVM

The remainder of this chapter explains more about these steps and components. For more coverage of Java in the Oracle database, you might also want to look at *Java Programming with Oracle JDBC* by Donald Bales (O'Reilly). For more comprehensive Java information, see the documentation from Sun Microsystems as well as the O'Reilly Java series (as well as several other books I'll recommend later in this chapter). For more detailed documentation on using Oracle and Java together, see Oracle Corporation's manuals.

Getting Ready to Use Java in Oracle

Before you can call Java methods from within your PL/SQL programs, you will need to do the following:

- Ensure that the Java option has been installed in your Oracle Database Server.
- Build and load your Java classes and code elements.
- In some cases, have certain Java-specific permissions granted to your Oracle user account.

Installing Java

On the Oracle server, the Java features may or may not be installed, depending on what version of Oracle you are running and what choices your DBA made during the Oracle installation. You can check whether Java is installed by running this query:

```
SELECT COUNT(*)
  FROM all_objects
 WHERE object_type LIKE 'JAVA%';
```

If the result is 0, Java is definitely not installed, and you can ask your DBA to run a script called *$ORACLE_HOME/javavm/install/initjvm.sql*.

As a developer, you will probably want to build and test Java programs on your own workstation, and that requires access to a Java Development Kit (JDK). You have two choices when installing the JDK: you can download it from *http://java.sun.com/* yourself; or, if you are using a third-party IDE such as Oracle JDeveloper, you may be able to rely on its bundled JDK. Be warned: you may need to be cautious about matching the exact JDK version number.

When you download Java from the Sun site, you will have to choose from among lots of different acronyms and versions. Personally, I've had reasonably good luck with Java 2 Standard Edition (J2SE) using the Core Java package rather than the "Desktop" package, the latter of which includes a bunch of GUI-building stuff I don't need. Another choice is between the JDK and the Java Runtime Engine (JRE). Always pick the JDK if you want to compile anything! In terms of the proper version to download, I would look at your Oracle server's version and try to match that.

Oracle version	JDK version
Oracle8i Database (8.1.5)	JDK 1.1.6
Oracle8i Database (8.1.6 or later)	JDK 1.2
Oracle9*i* Database	J2SE 1.3
Oracle Database 10*g* Release 1	J2SE 1.4.1
Oracle Database 10*g* Release 2	J2SE 1.4.2

If you have to support more than one version of the Oracle server, get the later one and be careful about what features you use.

One other unobvious thing you may need to know: if you can't seem to get your Java program to compile, check to see that the environment variable CLASSPATH has been set to include your classes—and the Oracle-supplied classes as well.

Building and Compiling Your Java Code

Many PL/SQL developers (myself included) have little experience with object-oriented languages, so getting up to speed on Java can be a bit of a challenge. In the short time in which I have studied and used Java, I have come to these conclusions:

- It doesn't take long to get a handle on the syntax needed to build simple classes in Java.
- It's not at all difficult to start leveraging Java inside PL/SQL, *but* ...
- Writing real object-oriented applications using Java requires significant learning and rethinking for PL/SQL developers!

There are many (many, many, many) books available on various aspects of Java, and a number of them are excellent. I recommend that you check out the following:

The Java Programming Language, by Ken Arnold, James Gosling, and David Holmes (Addison-Wesley)
James Gosling is the creator of Java, so you'd expect the book to be helpful. It is. Written in clear, simple terms, it gives you a strong grounding in the language.

Java in a Nutshell, by David Flanagan (O'Reilly)
This very popular and often-updated book contains a short, but excellent primer to the language, followed by a quick reference to all of the major language elements, arranged in an easy-to-use and heavily cross-referenced fashion.

Thinking in Java, by Bruce Eckel (Prentice Hall)
A very readable and creative approach to explaining object-oriented concepts. It is also available in a free, downloadable format at *http://www.mindview.net/Books/TIJ/*. If you like the feel of *Oracle PL/SQL Programming*, you will definitely enjoy *Thinking in Java*.

Later in this chapter, when I demonstrate how to call Java methods from within PL/SQL, I will also take you step by step through the creation of relatively simple classes. You will find that, in many cases, this discussion will be all you need to get the job done.

Setting Permissions for Java Development and Execution

Java security was handled differently prior to release 8.1.6 of the Oracle database, so we will look at the two models individually in the following sections.

Java security for Oracle through 8.1.5

Early releases of Oracle8*i* Database (before 8.1.6) supported a relatively simple model of Java security. There were basically two database roles that a DBA could grant:

JAVAUSERPRIV
: Grants relatively few Java permissions, including examining properties

JAVASYSPRIV
: Grants major permissions, including updating JVM-protected packages

So for example, if I want to allow Scott to perform any kind of Java-related operation, I would issue this command from a SYSDBA account:

```
GRANT JAVASYSPRIV TO scott;
```

If I want to place some restrictions on what he can do with Java, I might execute this grant instead:

```
GRANT JAVAUSERPRIV TO scott;
```

For example, to create a file through Java, I need the JAVASYSPRIV role; to read or write a file, I only need the JAVAUSERPRIV role. See Oracle's *Java Developer's Guide* from 8.1.7 (or later) for more details about which Java privileges correspond to which Oracle roles.

When the JVM is initialized, it installs an instance of java.lang.SecurityManager, the Java Security Manager. Oracle uses this, along with Oracle Database security, to determine who can call a particular Java method.

If a user lacking sufficient privileges tries to execute an illegal operation, the JVM will throw the java.lang.SecurityException. Here is what you would see in SQL*Plus:

```
ORA-29532: Java call terminated by uncaught Java exception:
           java.lang.SecurityException
```

When you run Java methods inside the database, different security issues can arise, particularly when interacting with the server-side filesystem or other operating system resources. Oracle follows the following two rules when checking I/O operations:

- If the dynamic ID has been granted JAVASYSPRIV, then Security Manager allows the operation to proceed.

- If the dynamic ID has been granted JAVAUSERPRIV, then Security Manager follows the same rules that apply to the PL/SQL UTL_FILE package to determine if the operation is valid. In other words, the file must be in a directory (or subdirectory) specified by the UTL_FILE_DIR parameter in the database initialization file.

Java security for Oracle from 8.1.6

Beginning with 8.1.6, Oracle's JVM offered support for Java 2 security, in which permissions are granted on a class-by-class basis. This is a much more sophisticated and fine-grained approach to security. This section offers some examples to give you a sense of the kind of security-related code you could write (check Oracle's manuals for more details and examples).

Generally, you will use the DBMS_JAVA.GRANT_PERMISSION procedure to grant the appropriate permissions. Here is an example of calling that program to give the BATCH schema permission to read and write the *lastorder.log* file:

```
/* Must be connected as a dba */
BEGIN
   DBMS_JAVA.grant_permission(
      grantee => 'BATCH',
      permission_type => 'java.io.FilePermission',
      permission_name => '/apps/OE/lastorder.log',
      permission_action => 'read,write');
END;
/
COMMIT;
```

When making such a call, be sure to uppercase the grantee; otherwise, Oracle won't be able to locate the account name.

Also note the COMMIT. It turns out that this DBMS_JAVA call is just writing permission data to a table in Oracle's data dictionary, but it does not commit automatically. And, by the way, you can query permission data through the views USER_JAVA_POLICY and DBA_JAVA_POLICY.

Here is a sequence of commands that first grants permission to access files in a directory, and then restricts permissions on a particular file:

```
BEGIN
/* First, grant read and write to everyone */
   DBMS_JAVA.grant_permission(
      'PUBLIC',
      'java.io.FilePermission',
      '/shared/*',
      'read,write');

/* Use the "restrict" built-in to revoke read & write
|  permission on one particular file from everyone
*/
   DBMS_JAVA.restrict_permission(
      'PUBLIC',
      'java.io.FilePermission',
      '/shared/secretfile',
      'read,write');

/* Now override the restriction so that one user can read and write
|  that file.
*/
   DBMS_JAVA.grant_permission(
      'BOB',
      'java.io.FilePermission',
      '/shared/secretfile',
      'read,write');

  COMMIT;
END;
```

Here are the predefined permissions that Oracle offers:

```
java.util.PropertyPermission
java.io.SerializablePermission
java.io.FilePermission
java.net.NetPermission
java.net.SocketPermission
java.lang.RuntimePermission
java.lang.reflect.ReflectPermission
java.security.SecurityPermission
oracle.aurora.rdbms.security.PolicyTablePermission
oracle.aurora.security.JServerPermission
```

Oracle also supports the Java mechanisms for creating your own permissions; check Oracle's *Java Developer's Guide* for details.

A Simple Demonstration

Before diving into the details, let's walk through all the steps needed to access Java from within PL/SQL. In the process, I'll introduce the various pieces of technology you need to get the job done.

Say that I need to be able to delete a file from within PL/SQL. Prior to Oracle8*i* Database, I had the following options.

- In Oracle7 Database (7.3) (and above), I could send a message to a database pipe and then have a C listener program grab the message ("Delete file X") and do all the work.
- In Oracle8 Database and later, I could set up a library that pointed to a C DLL or shared library, and then from within PL/SQL, call a program in that library to delete the file.

The pipe technique is handy, but it is a clumsy workaround. The external procedure implementation in Oracle8 Database is a better solution, but it is also less than straightforward, especially if you don't know the C language. So the Java solution looks as if it might be the best one all around. Although some basic knowledge of Java is required, you don't need the same level of skill that would be required to write the equivalent code in C. Java comes with prebuilt (foundation) classes that offer clean, easy-to-use APIs to a wide array of functionality, including file I/O.

Here are the steps that I will perform in this demonstration:

1. Identify the Java functionality I need to access.
2. Build a class of my own to make the underlying Java feature callable through PL/SQL.
3. Compile the class and load it into the database.
4. Build a PL/SQL program to call the class method I created.
5. Delete files from within PL/SQL.

Finding the Java Functionality

A while back, my O'Reilly editor, Deborah Russell, was kind enough to send me a whole bunch of their Java books, so I grabbed the big, fat *Java Fundamental Classes Reference*, by Mark Grand and Jonathan Knudsen, and looked up "File" in the index (sure, I could use online documentation, but I *like* books). The entry for "File class" caught my eye, and I hurried to the correct page.

There I found information about the class named java.io.File, namely, that it "provides a set of methods to obtain information about files and directories." And it doesn't just let you obtain information; it also contains methods (procedures and functions) to delete and rename files, make directories, and so on. I had come to the right place!

Here is a portion of the API offered by the File class:

```
public class java.io.File {
   public boolean delete();
   public boolean mkdir ();
}
```

In other words, I will call a Boolean function in Java to delete a file. If the file is deleted, the function returns TRUE; otherwise, it returns FALSE.

Building a Custom Java Class

Now, you might be asking yourself why I had to build my own Java class on top of the File class. Why can't I just call that function directly inside my PL/SQL wrapper? There are two reasons:

- A Java class method is typically executed for a specific object instantiated from the class. In PL/SQL I cannot instantiate a Java object and then call the method against that object; in other words, PL/SQL only allows the calling of *static* methods.

- Even though Java and PL/SQL both have Boolean datatypes (Java even offers a Boolean primitive and a Boolean class), they do not map to each other. I cannot pass a Boolean from Java directly to a PL/SQL Boolean.

As a direct consequence, I need to build my own class that will:

- Instantiate an object from the File class
- Execute the delete method against that object
- Return a value that PL/SQL interprets properly

Here is the very simple class I wrote to take advantage of the File.delete method:

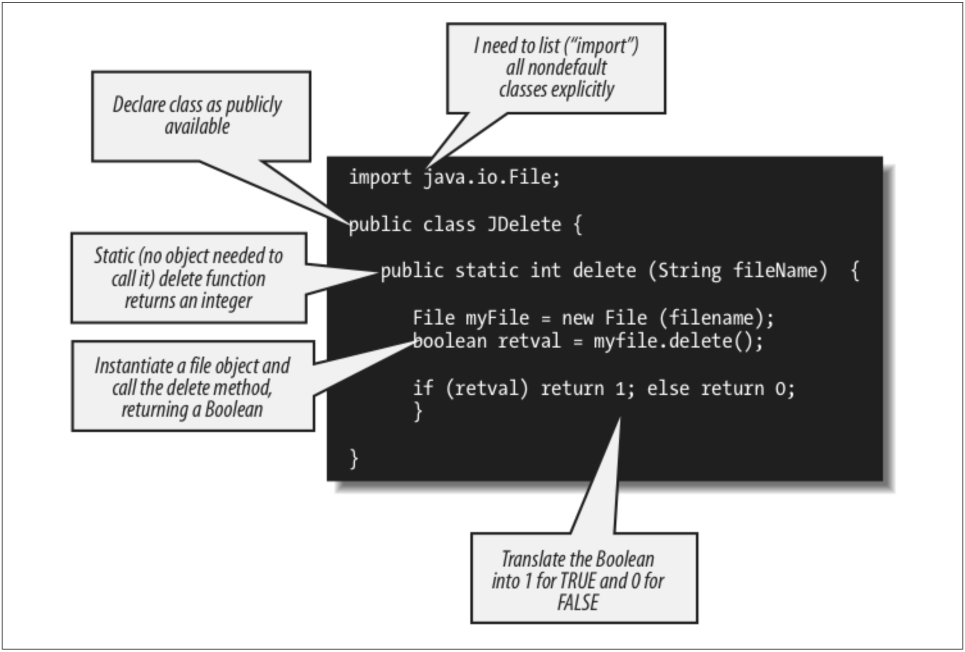
```
/* File on web: JDelete.java */
import java.io.File;

public class JDelete {

   public static int delete (String fileName) {
      File myFile = new File (fileName);
      boolean retval = myFile.delete();
      if (retval) return 1; else return 0;
   }
}
```

Figure 27-2 explains each of the steps in this code, but the main effect is clear: the JDelete.delete method simply instantiates a dummy File object for the specified filename so that I can call the delete method for that file. By declaring my method to be static, I make that method available without the need to instantiate an object. Static methods are associated with the *class*, not with the individual instances of the objects of that class.

The JDelete class above highlights a number of differences between Java and PL/SQL that you should keep in mind:

- There are no BEGIN and END statements in Java for blocks, loops, or conditional statements. Instead, you use curly braces to delimit the block.
- Java is case-sensitive; "if " is definitely not the same thing as "IF".
- The assignment operator is a plain equals sign (=) rather than the compound symbol used in PL/SQL (:=).
- When you call a method that does not have any arguments (such as the delete method of the File class), you still must open and close the parentheses. Otherwise, the Java compiler will try to interpret the method as a class member or data structure.

Hey, that was easy! Of course, you didn't watch me fumble around with Java for a day, getting over the nuisance of minor syntax errors, the agony of a case-sensitive language, and the confusion setting the CLASSPATH. I'll leave all that to your imagination—and your own day of fumbling!


I need to list ("import") all nondefault classes explicitly
Declare class as publicly available
Static (no object needed to call it) delete function returns an integer
Instantiate a file object and call the delete method, returning a Boolean
Translate the Boolean into 1 for TRUE and 0 for FALSE

```
import java.io.File;

public class JDelete {

   public static int delete (String fileName)  {

      File myFile = new File (filename);
      boolean retval = myfile.delete();

      if (retval) return 1; else return 0;
      }

}
```

Figure 27-2. A simple Java class used to delete a file

Compiling and Loading into Oracle

Now that my class is written, I need to compile it. On a Microsoft Windows machine, one way I could do this would be to open a console session on the directory where I have the source code, ensure that the Java compiler (*javac.exe*) is on my PATH, and do this:

```
C:\samples\java>javac JDelete.java
```

If successful, the compiler should generate a file called JDelete.class.

Now that it's compiled, I realize that it would make an awful lot of sense to test the function before I stick it inside Oracle and try it from PL/SQL. You are always better off building and testing *incrementally*. Java gives us an easy way to do this: the "main" method. If you provide a void method (i.e., a procedure) called main in your class—and give it the right parameter list—you can then call the class, and this code will execute.

The main method is one example of how Java treats certain elements in a special way if they have the right signature. Another example is the toString method. If you add a method with this name to your class, it will automatically be called to display your custom description of the object. This is especially useful when your object consists of many elements that make sense only when presented a certain way, or that otherwise require formatting to be readable.

So let's add a simple main method to JDelete:

```
import java.io.File;

public class JDelete {
   public static int delete (String fileName) {
      File myFile = new File (fileName);
      boolean retval = myFile.delete();
      if (retval) return 1; else return 0;
      }

   public static void main (String args[]) {
      System.out.println (
         delete (args[0])
         );
   }
}
```

The first element (0) in the "args" array represents the first argument supplied from the calling environment.

Next, I will recompile the class:

```
C:\samples\java>javac JDelete.java
```

And, assuming the "java" executable is on my PATH:

```
C:\samples\java>java JDelete c:\temp\te_employee.pks
1

C:\samples\java>java JDelete c:\temp\te_employee.pks
0
```

Notice that the first time I run the main method it displays 1 (TRUE), indicating that the file was deleted. So it will come as no surprise that when I run the same command again, main displays 0. It couldn't delete a file that had already been deleted.

That didn't take too much work or know-how, did it?

In another demonstration of the superiority of Java over PL/SQL, please note that while you have to type 20 characters in PL/SQL to display output (DBMS_OUTPUT.PUT_LINE), you needn't type any more than 18 characters in Java (System.out.println). Give us a break, you language designers! Though Alex Romankeuich, one of our technical reviewers, notes that if you declare "private static final PrintStream o = System.out;" at the beginning of the class, you can then display output in the class with the command "o.println"—only 9 characters in all!

Now that my class compiles, and I have verified that the delete method works, I can load it into the SCOTT schema of the Oracle database using Oracle's *loadjava* command. Oracle includes *loadjava* as part of its distribution, so it should be on your PATH if you have installed the Oracle server or client on your local machine.

```
C:\samples\java>loadjava -user scott/tiger -oci8 -resolve JDelete.class
```

I can even verify that the class is loaded by querying the contents of the USER_OBJECTS data dictionary via a utility I'll introduce later in this chapter:

```
SQL> EXEC myjava.showobjects
Object Name                  Object Type   Status  Timestamp
---------------------------------------------------------------------
JDelete                      JAVA CLASS    VALID   2005-05-06:15:01
```

That takes care of all the Java-specific steps, which means that it's time to return to the cozy world of PL/SQL.

Building a PL/SQL Wrapper

I will now make it easy for anyone connecting to my database to delete files from within PL/SQL. To accomplish this goal, I will create a PL/SQL wrapper that looks like a PL/SQL function on the outside but is really nothing more than a pass-through to the underlying Java code.

```
/* File on web: fdelete.sf */
FUNCTION fDelete (
   file IN VARCHAR2)
   RETURN NUMBER
AS LANGUAGE JAVA
   NAME 'JDelete.delete (
         java.lang.String)
         return int';
```

The implementation of the fdelete function consists of a string describing the Java method invocation. The parameter list must reflect the parameters of the method, but in place of each parameter, I specify the fully qualified datatype name. In this case, that means that I cannot simply say "String," but instead must add the full name of the Java package containing the String class. The RETURN clause simply lists int for integer. The int is a primitive datatype, not a class, so that is the complete specification.

As a bit of an aside, I could also write a call spec for a procedure that invokes the JDelete.main method:

```
PROCEDURE fDelete2 (
   file IN VARCHAR2)
AS LANGUAGE JAVA
   NAME 'JDelete.main(java.lang.String[])';
```

The main method is special; even though it accepts an array of Strings, you can define a call spec using any number of parameters.

Deleting Files from PL/SQL

So I compile the function and then prepare to perform my magical, previously difficult (if not impossible) feat:

```
SQL> @fdelete.sf

Function created.

SQL> EXEC DBMS_OUTPUT.PUT_LINE (fdelete('c:\temp\te_employee.pkb'))
```

And I get:

```
ERROR at line 1:
ORA-29532: Java call terminated by uncaught Java exception: java.security.
AccessControlException: the
Permission (java.io.FilePermission c:\temp\te_employee.pkb delete) has not been
granted to BOB. The PL/SQL
to grant this is dbms_java.grant_permission( 'BOB', 'SYS:java.io.FilePermission',
'c:\temp\te_employee.pkb', 'delete' )
ORA-06512: at "BOB.FDELETE", line 1
ORA-06512: at line 1
```

I forgot to give myself permission! But hey, look at that message—it's pretty nice of Oracle to tell me not just what the problem is but also how to fix it. So I get my friendly DBA to run something like this (a slight variation of Oracle's suggestion):

```
CALL DBMS_JAVA.grant_permission(
   'BOB',
   'SYS:java.io.FilePermission',
   'c:\temp\*',
   'read,write,delete' );
```

And now I get:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE (fdelete('c:\temp\te_employee.pkb'))
1
SQL> exec DBMS_OUTPUT.PUT_LINE (fdelete('c:\temp\te_employee.pkb'))
0
```

Yippee, it works!

I can also build utilities on top of this function. How about a procedure that deletes all of the files found in the rows of a nested table? Even better, how about a procedure that accepts a directory name and filter ("all files like *.*tmp*," for example) and deletes all files found in that directory that pass the filter?

In reality, of course, what I should do is build a package and then put all this great new stuff in there. And that is just what I will do later in this chapter. First, however, let's take a closer look at each of the steps I just performed.

Using loadjava

The *loadjava* utility is an operating system command-line utility that uploads Java files into the database. The first time you run *loadjava* in a schema, it creates a number of objects in your local schema; although the exact list varies somewhat by Oracle version, you are likely to find at least the following in your default tablespace:

CREATE$JAVA$LOB$TABLE
: A table created in each schema, containing Java code elements. Each new class you load using *loadjava* will generate one row in this table, putting the bytes of the class into a BLOB column.

SYS_C ... (exact name will vary)
: Unique index on the above table.

SYS_IL ... (exact name will vary)
: LOB index on the above table.

By the way, if you don't have permissions or quota available to create these objects in your default tablespace, the load operation will fail.

Before executing the load, Oracle will check to see if the object being loaded already exists, and whether it has changed, thereby minimizing the need to reload and avoiding invalidation of dependent classes.[*]

The load operation then calls the DDL command CREATE JAVA to load the Java classes from the BLOB column of CREATE$JAVA$LOB$TABLE into the RDBMS as schema objects. This loading occurs only if:

- The class is being loaded for the first time.
- The class has been changed.
- The `-force` option is supplied.

Here is the basic syntax :

```
loadjava {-user | -u} username/password[@database]
  [option ...] filename [filename ]...
```

where *filename* is a Java source, sqlj, class, *.jar*, resource, properties, or *.zip* file. The following command, for example, loads the JFile class into the SCOTT schema:

[*] Oracle examines the MD5 checksum of the incoming class and compares it against that of the existing class.

```
C:> loadjava -user scott/tiger -oci8 -resolve JFile.class
```

Here are some things to keep in mind about *loadjava*. To display help text, use this syntax:

```
loadjava {-help | -h}
```

In a list of options or files, names must be separated only by spaces:

```
-force, -resolve, -thin  // No
-force -resolve -thin    // Yes
```

In a list of users or roles, however, names must be separated only by commas:

```
SCOTT, PAYROLL, BLAKE  // No
SCOTT,PAYROLL,BLAKE    // Yes
```

There are more than 40 command-line options on *loadjava*; some key options are mentioned in Table 27-2.

Table 27-2. Common loadjava options

Option	Description
-debug	Aids in debugging the *loadjava* script itself, not your code; rarely necessary.
-definer	Specifies that the methods of uploaded classes will execute with the privileges of their definer, not their invoker. (By default, methods execute with the privileges of their invoker.) Different definers can have different privileges, and an application can have many classes, so programmers should make sure the methods of a given class execute only with the privileges they need.
-encoding	Sets (or resets) the -encoding option in the database table JAVA$OPTIONS to the specified value, which must be the name of a standard JDK encoding scheme (the default is "latin1"). The compiler uses this value, so the encoding of uploaded source files must match the specified encoding. Refer to the section "GET_, SET_, and RESET_COMPILER_OPTION: Getting and Setting (a Few) Compiler Options" on page 1151 for information on how this object is created and used.
-force	Forces the loading of Java class files, whether or not they have changed since they were last loaded. Note that you cannot force the loading of a class file if you previously loaded the source file (or vice versa). You must drop the originally loaded object first.
-grant	Grants the EXECUTE privilege on uploaded classes to the listed users or roles. (To call the methods of a class directly, users must have the EXECUTE privilege.) This option is cumulative. Users and roles are added to the list of those having the EXECUTE privilege. To revoke the privilege, either drop and reload the schema object without specifying -grant, or use the SQL REVOKE statement. To grant the privilege on an object in another user's schema, you must have the CREATE PROCEDURE WITH GRANT privilege.
-oci8	Directs *loadjava* to communicate with the database using the OCI JDBC driver. This option (the default) and -thin are mutually exclusive. When calling *loadjava* from a client-side computer that does not have Oracle installed on it, use the -thin option.
-resolve	After all files on the command line are loaded and compiled (if necessary), resolves all external references in those classes. If this option is not specified, files are loaded but not compiled or resolved until runtime.

Option	Description
	Specify this option to compile and resolve a class that was loaded previously. You need not specify the -force option because resolution is done independently, after loading.
-resolver	Binds newly created class schema objects to a user-defined resolver spec. Because it contains spaces, the resolver spec must be enclosed in double quotes. This option and -oracleresolver (the default) are mutually exclusive.
-schema	Assigns newly created Java schema objects to the specified schema. If this option is not specified, the logon schema is used. You must have the CREATE ANY PROCEDURE privilege to load into another user's schema.
-synonym	Creates a public synonym for uploaded classes, making them accessible outside the schema into which they are loaded. To specify this option, you must have the CREATE PUBLIC SYNONYM privilege. If you specify this option for source files, it also applies to classes compiled from those source files.
-thin	Directs *loadjava* to communicate with the database using the thin JDBC driver. This option and -oci8 (the default) are mutually exclusive. When calling *loadjava* from a client-side computer that does not have Oracle installed on it, use the -thin option.
-verbose	Enables the verbose mode, in which progress messages are displayed. Very handy!

As you can probably imagine, there are various nuances of using *loadjava*, such as whether to load individual classes or compressed groups of elements in a *.zip* or *.jar* file. The Oracle documentation contains more information about the *loadjava* command.

Using dropjava

The *dropjava* utility reverses the action of *loadjava*. It converts filenames into the names of schema objects, then drops the schema objects and any associated data. Dropping a class invalidates classes that depend on it directly or indirectly. Dropping a source object also drops classes derived from it.

The syntax is nearly identical to *loadjava* syntax:

```
dropjava {-user | -u} username/password[@database]
  [option ...] filename [filename] ...
```

where *option* includes -oci8, -encoding, and -verbose.

Managing Java in the Database

This section explores in more detail issues related to the way that Java elements are stored in the database and how you can manage those elements.

The Java Namespace in Oracle

Oracle stores each Java class in the database as a schema object. The name of that object is derived from (but is not the same as) the fully qualified name of the class; this name includes the names of any containing packages. The full name of the class OracleSimpleChecker, for example, is as follows:

```
oracle.sqlj.checker.OracleSimpleChecker
```

In the database, however, the full name of the Java schema object would be:

```
oracle/sqlj/checker/OracleSimpleChecker
```

In other words, once stored in the Oracle database, slashes replace dots.

An object name in Oracle, whether the name of a database table or a Java class, cannot be longer than 30 characters. Java does not have this restriction; you can have much longer names. Oracle will allow you to load a Java class into the Oracle database with a name of up to 4,000 characters. If the Java element name has more than 30 characters, the database will automatically generate a valid alias (less than 31 characters) for that element.

But don't worry! You never have to reference that alias in your stored procedures. You can instead continue to use the real name for your Java element in your code. Oracle will map that long name automatically to its alias (the schema name) when necessary.

Examining Loaded Java Elements

Once you have loaded Java source, class, and resource elements into the database, information about those elements is available in several different data dictionary views, as shown in Table 27-3.

Table 27-3. Class information in data dictionary views

View	Description
USER_OBJECTS	Contains header information about your objects of JAVA SOURCE, JAVA CLASS, and JAVA RESOURCE types
ALL_OBJECTS	
DBA_OBJECTS	
USER_ERRORS	Contains any compilation errors encountered for your objects
ALL_ERRORS	
DBA_ERRORS	
USER_SOURCE	Contains the source code for your Java source if you used the CREATE JAVA SOURCE command to create the Java schema object

Here is a query that shows all of the Java-related objects in my schema:

```
/* Files on web: showjava.sql, myJava.pkg */
COLUMN object_name FORMAT A30
SELECT object_name, object_type, status, timestamp
  FROM user_objects
 WHERE (object_name NOT LIKE 'SYS_%'
        AND object_name NOT LIKE 'CREATE$%'
```

```
         AND object_name NOT LIKE 'JAVA$%'
         AND object_name NOT LIKE 'LOADLOB%')
   AND object_type LIKE 'JAVA %'
  ORDER BY object_type, object_name;
```

The WHERE clause filters out those objects created by Oracle for managing Java objects. You can build programs to access the information in a variety of useful ways. Here is some sample output from the myjava package, which you can find on the book's web site:

```
SQL> EXEC myJava.showObjects
Object Name                    Object Type   Status  Timestamp
---------------------------------------------------------------------
DeleteFile                     JAVA CLASS    INVALID 0000-00-00:00:00
JDelete                        JAVA CLASS    VALID   2005-05-06:10:13
book                           JAVA CLASS    VALID   2005-05-06:10:07
DeleteFile                     JAVA SOURCE   INVALID 2005-05-06:10:06
book                           JAVA SOURCE   VALID   2005-05-06:10:07
```

The following would let you see a list of all the Java elements whose names contain "Delete":

```
SQL> EXEC myJava.showobjects ('%Delete%')
```

The column USER_OBJECTS.object_name contains the full names of Java schema objects, unless the name is longer than 30 characters or contains an untranslatable character from the Unicode character set. In both cases, the short name is displayed in the object_name column. To convert short names to full names, you can use the LONGNAME function in the utility package DBMS_JAVA, which is explored in the next section.

Using DBMS_JAVA

The Oracle built-in package DBMS_JAVA gives you access to and the ability to modify various characteristics of the Java Virtual Machine in the database.

The DBMS_JAVA package contains a large number of programs, many of which are intended for Oracle internal use only. Nevertheless, we can take advantage of a number of very useful programs; most can also be called within SQL statements. Table 27-4 summarizes some of the DBMS_JAVA programs. As noted earlier in the chapter, DBMS_JAVA also offers programs to manage security and permissions.

Table 27-4. Common DBMS_JAVA programs

Program	Description
LONGNAME function	Obtains the full (long) Java name for a given Oracle short name
GET_COMPILER_OPTION function	Looks up an option in the Java options table
SET_COMPILER_OPTION procedure	Sets a value in the Java options table and creates the table, if one does not exist
RESET_COMPILER_OPTION procedure	Resets a compiler option in the Java options table

Program	Description
SET_OUTPUT procedure	Redirects Java output to the DBMS_OUTPUT text buffer
EXPORT_SOURCE procedure	Exports a Java source schema object into an Oracle LOB
EXPORT_RESOURCE procedure	Exports a Java resource schema object into an Oracle LOB
EXPORT_CLASS procedure	Exports a Java class schema object into an Oracle LOB

These programs are explored in more detail in the following sections.

LONGNAME: Converting Java Long Names

Java class names can easily exceed the maximum SQL identifier length of 30 characters. In such cases, Oracle creates a unique "short name" for the Java code element and uses that name for SQL- and PL/SQL-related access.

Use the following function to obtain the full (long) name for a given short name:

```
FUNCTION DBMS_JAVA.LONGNAME (shortname VARCHAR2) RETURN VARCHAR2
```

The following query displays the long names for all Java classes defined in the currently connected schema for which the long names and short names do not match:

```
/* File on web: longname.sql */
SELECT object_name shortname,
       DBMS_JAVA.LONGNAME (object_name) longname
  FROM USER_OBJECTS
  WHERE object_type = 'JAVA CLASS'
    AND object_name != DBMS_JAVA.LONGNAME (object_name);
```

This query is also available inside the myJava package (found in the *myJava.pkg* file); its use is shown here. Suppose that I define a class with this name:

```
public class DropAnyObjectIdentifiedByTypeAndName {
```

That is too long for Oracle, and I can verify that Oracle creates its own short name as follows:

```
SQL> EXEC myJava.showlongnames
Short Name | Long Name
----------------------------------------------------
Short: /247421b0_DropAnyObjectIdentif
Long:  DropAnyObjectIdentifiedByTypeAndName
```

GET_, SET_, and RESET_COMPILER_OPTION: Getting and Setting (a Few) Compiler Options

You can also set a few of the compiler option values in the database table JAVA$OPTIONS (called the *options table* from here on). While there are currently about 40 command-line options, only three of them can be saved in the options table. They are:

encoding
: Character-set encoding in which the source code is expressed. If not specified, the compiler uses a default, which is the result of the Java method System.getProperty("file.encoding"); a sample value is `ISO646-US`.

online
: True or false; applies only to SQLJ source. The default value of "true" enables online semantics checking.

debug
: True or false; setting to true is like using javac -g. If not specified, the compiler defaults to true.

The compiler looks up options in the options table unless they are specified on the *loadjava* command line.

You can get and set these three options-table entries using the following DBMS_JAVA functions and procedures:

```
FUNCTION DBMS_JAVA.GET_COMPILER_OPTION (
   what VARCHAR2, optionName VARCHAR2)

PROCEDURE DBMS_JAVA.SET_COMPILER_OPTION (
   what VARCHAR2, optionName VARCHAR2, value VARCHAR2)

PROCEDURE DBMS_JAVA.RESET_COMPILER_OPTION (
   what VARCHAR2, optionName VARCHAR2)
```

where:

what
: Is the name of a Java package, the full name of a class, or the empty string. After searching the options table, the compiler selects the row in which *what* most closely matches the full name of the schema object. If *what* is the empty string, it matches the name of any schema object.

optionName
: Is the name of the option being set. Initially, a schema does not have an options table. To create one, use the procedure DBMS_JAVA.SET_COMPILER_OPTION to set a *value*. The procedure creates the table if it does not exist. Enclose parameters in single quotes, as shown in the following example:

```
SQL> DBMS_JAVA.SET_COMPILER_OPTION ('X.sqlj', 'online', 'false');
```

SET_OUTPUT: Enabling Output from Java

When executed within the Oracle database, the System.out and System.err classes send their output to the current trace files, which are typically found in the server's *udump* subdirectory. This is not a very convenient location if you simply want to test your code to see if it is working properly. DBMS_JAVA supplies a procedure you can call to redirect output to the DBMS_OUTPUT text buffer so that it can be flushed to your SQL*Plus screen automatically. The syntax of this procedure is:

```
PROCEDURE DBMS_JAVA.SET_OUTPUT (buffersize NUMBER);
```

Here is an example of how you might use this program:

```
/* File on web: ssoo.sql */
SET SERVEROUTPUT ON SIZE 1000000
CALL DBMS_JAVA.SET_OUTPUT (1000000);
```

Passing any integer to DBMS_JAVA.set_output will turn it on. Documentation on the interaction between these two commands is skimpy; my testing has uncovered the following behaviors:

- The minimum (and default) buffer size is a measly 2,000 bytes; the maximum size is 1,000,000 bytes, at least up through Oracle Database 10*g* Release 1. You can pass a number outside of that range without causing an error; unless the number is *really* big, the maximum will be set to 1,000,000.
- The buffer size specified by SET SERVEROUTPUT supersedes that of DBMS_JAVA.SET_OUTPUT. In other words, if you provide a smaller value for the DBMS_JAVA call, it will be ignored, and the larger size will be used.
- If you are running Oracle Database 10g Release 2 or later, and you have set SERVEROUTPUT size to be UNLIMITED, the maximum size of the Java buffer is also unlimited.
- If your output in Java exceeds the buffer size, you will *not* receive the error you get with DBMS_OUTPUT, namely:

  ```
  ORA-10027: buffer overflow, limit of nnn bytes
  ```

  The output will instead be truncated to the buffer size specified, and execution of your code will continue.

As is the case with DBMS_OUTPUT, you will not see any output from your Java calls until the stored procedure through which they are called finishes executing.

EXPORT_SOURCE, EXPORT_RESOURCE, and EXPORT_CLASS: Exporting Schema Objects

Oracle's DBMS_JAVA package offers the following set of procedures to export source, resources, and classes:

```
PROCEDURE DBMS_JAVA.EXPORT_SOURCE (
   name VARCHAR2 IN,
   [ blob BLOB IN | clob CLOB IN ]
   );

PROCEDURE DBMS_JAVA.EXPORT_SOURCE (
   name VARCHAR2 IN,
   schema VARCHAR2 IN,
```

```
      [ blob BLOB IN | clob CLOB IN ]
      );

   PROCEDURE DBMS_JAVA.EXPORT_RESOURCE (
      name VARCHAR2 IN,
      [ blob BLOB IN | clob CLOB IN ]
      );

   PROCEDURE DBMS_JAVA.EXPORT_RESOURCE (
      name VARCHAR2 IN,
      schema VARCHAR2 IN,
      [ blob BLOB IN | clob CLOB IN ]
      );

   PROCEDURE DBMS_JAVA.EXPORT_CLASS (
      name VARCHAR2 IN,
      blob BLOB IN
      );

   PROCEDURE DBMS_JAVA.EXPORT_CLASS (
      name VARCHAR2 IN,
      schema VARCHAR2 IN,
      blob BLOB IN
      );
```

In all cases, *name* is the name of the Java schema object to be exported, *schema* is the name of the schema owning the object (if one is not supplied, the current schema is used), and *blob* | *clob* is the large object that receives the specified Java schema object.

You cannot export a class into a CLOB, only into a BLOB. In addition, the internal representation of the source uses the UTF8 format, so that format is used to store the source in the BLOB as well.

The following prototype procedure offers an idea of how you might use the export programs to obtain source code of your Java schema objects, when appropriate:

```
/* File on web: showjava.sp */
PROCEDURE show_java_source (
   NAME IN VARCHAR2, SCHEMA IN VARCHAR2 := NULL
)
-- Overview: Shows Java source (prototype). Author: Vadim Loevski
IS
   b                      CLOB;
   v                      VARCHAR2 (2000);
   i                      INTEGER;
   object_not_available   EXCEPTION;
   PRAGMA EXCEPTION_INIT(object_not_available, -29532);

BEGIN
   /* Move the Java source code to a CLOB. */
   DBMS_LOB.createtemporary(b, FALSE );

   DBMS_JAVA.export_source(name,  NVL(SCHEMA, USER), b);

   /* Read the CLOB to a VARCHAR2 variable and display it. */
```

```
      i := 1000;
      DBMS_lob.read(b, i, 1, v);
      DBMS_OUTPUT.put_line(v);
   EXCEPTION
      /* If the named object does not exist, an exception is raised. */
      WHEN object_not_available
      THEN
         IF DBMS_UTILITY.FORMAT_ERROR_STACK LIKE '%no such%object'
         THEN
            DBMS_OUTPUT.put_line ('Java object cannot be found.' );
         END IF;
   END;
```

If I then create a Java source object using the CREATE JAVA statement as follows:

```
   CREATE OR REPLACE JAVA SOURCE NAMED "Hello"
   AS
      public class Hello {
         public static String hello() {
            return "Hello Oracle World";
         }
            };
```

I can view the source code as shown here (assuming that DBMS_OUTPUT has been enabled):

```
   SQL> EXEC show_java_source ('Hello')
   public class Hello {
         public static String hello() {
            return "Hello Oracle World";
         }
   };
```

Publishing and Using Java in PL/SQL

Once you have written your Java classes and loaded them into the Oracle database, you can call their methods from within PL/SQL (and SQL)—but only after you "publish" those methods via a PL/SQL wrapper.

Call Specs

You need to build wrappers in PL/SQL only for those Java methods you want to make available through a PL/SQL interface. Java methods can access other Java methods in the Java Virtual Machine directly, without any need for a wrapper. To publish a Java method, you write a *call spec*—a PL/SQL program header (function or procedure) whose body is actually a call to a Java method via the LANGUAGE JAVA clause. This clause contains the following information about the Java method: its full name, its parameter types, and its return type. You can define these call specs as standalone functions or procedures, as programs within a package, or as methods in an object type:

```
   CREATE [OR REPLACE] --If standalone (not in a package)
   Standard PL/SQL procedure/function header
```

```
{IS | AS} LANGUAGE JAVA
NAME 'method_fullname (java_type[, java_type]...)
  [return java_type]';
```

where *java_type* is either the full name of a Java type, such as java.lang.String, or a primitive type, such as int. Note that you do *not* include the parameter names, only their types.

The NAME clause string uniquely identifies the Java method being wrapped. The full Java name and the call spec parameters, which are mapped by position, must correspond, one to one, with the parameters in the program. If the Java method takes no arguments, code an empty parameter list for it.

Here are a few examples:

- As shown earlier, here is a standalone function calling a method:

```
FUNCTION fDelete (
   file IN VARCHAR2)
   RETURN NUMBER
AS LANGUAGE JAVA
   NAME 'JDelete.delete (
         java.lang.String)
         return int';
```

- A packaged procedure that passes an object type as a parameter:

```
PACKAGE nat_health_care
IS
   PROCEDURE consolidate_insurer (ins Insurer)
      AS LANGUAGE JAVA
      NAME 'NHC_consolidation.process(oracle.sql.STRUCT)';
END nat_health_care;
```

- An object type method:

```
TYPE pet_t AS OBJECT (
  name VARCHAR2(100),
  MEMBER FUNCTION date_of_birth (
    name_in IN VARCHAR2) RETURN DATE
    AS LANGUAGE JAVA
    NAME 'petInfo.dob (java.lang.String)
             return java.sql.Timestamp'
);
```

- A standalone procedure with an OUT parameter:

```
PROCEDURE read_out_file (
   file_name   IN      VARCHAR2,
   file_line   OUT     VARCHAR2
)
AS
LANGUAGE JAVA
   NAME 'utils.ReadFile.read(java.lang.String
                      ,java.lang.String[])';
```

Some Rules for Call Specs

Note the following:

- A PL/SQL call spec and the Java method it publishes must reside in the same schema.
- A call spec exposes a Java method's top-level entry point to Oracle. As a result, you can publish only public static methods, unless you are defining a member method of a SQL object type. In this case, you can publish instance methods as member methods of that type.
- You cannot provide default values in the parameter list of the PL/SQL program that will serve as a wrapper for a Java method invocation.
- A method in object-oriented languages cannot assign values to objects passed as arguments; the point of the method is to apply to the object to which it is attached. When you want to call a method from SQL or PL/SQL and change the value of an argument, you must declare it as an OUT or IN OUT parameter in the call spec. The corresponding Java parameter must then be a one-element array of the appropriate type.

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

A function that declares OUT or IN OUT parameters cannot be called from SQL DML statements.

Mapping Datatypes

Earlier in this chapter, we saw a very simple example of a PL/SQL wrapper—a delete function that passed a VARCHAR2 value to a java.lang.String parameter. The Java method returned an int, which was then passed back through the RETURN NUMBER clause of the PL/SQL function. These are straightforward examples of datatype mapping, that is, setting up a correspondence between a PL/SQL datatype and a Java datatype.

When you build a PL/SQL call spec, the PL/SQL and Java parameters, as well as the function result, are related by position and must have compatible datatypes. Table 27-5 lists all the datatype mappings currently allowed between PL/SQL and Java. If you rely on a supported datatype mapping, Oracle will convert from one to the other automatically.

Table 27-5. Legal datatype mappings

SQL type	Java class
CHAR	oracle.sql.CHAR
NCHAR	java.lang.String
LONG	java.sql.Date
VARCHAR2	java.sql.Time
NVARCHAR2	java.sql.Timestamp
	java.lang.Byte
	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	byte, short, int, long, float, double
DATE	oracle.sql.DATE
	java.sql.Date
	java.sql.Time
	java.sql.Timestamp
	java.lang.String
NUMBER	oracle.sql.NUMBER
	java.lang.Byte
	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	byte, short, int, long, float, double
RAW	oracle.sql.RAW
LONG RAW	byte[]
ROWID	oracle.sql.CHAR
	oracle.sql.ROWID
	java.lang.String

SQL type	Java class
BFILE	oracle.sql.BFILE
BLOB	oracle.sql.BLOB
	oracle.jdbc2.Blob
CLOB	oracle.sql.CLOB
NCLOB	oracle.jdbc2.Clob
User-defined object type	oracle.sql.STRUCT
	java.sql.Struct
	java.sql.SqlData
	oracle.sql.ORAData
User-defined REF type	oracle.sql.REF
	java.sql.Ref
	oracle.sql.ORAData
Opaque type (such as XMLType)	oracle.sql.OPAQUE
TABLE	oracle.sql.ARRAY
VARRAY	
User-defined table or VARRAY type	java.sql.Array
	oracle.sql.ORAData
Any of the above SQL types	oracle.sql.CustomDatum
	oracle.sql.Datum

As you can see, Oracle supports only automatic conversion for SQL datatypes. Such PL/SQL-specific datatypes as BINARY_INTEGER, PLS_INTEGER, BOOLEAN, and associative array types are not supported. In those cases, you have to perform manual conversion steps to transfer data between these two execution environments. See the references in the section "Other Examples" on page 1167 for examples of nondefault mappings; see the Oracle documentation for even more detailed examples involving the use of JDBC.

Calling a Java Method in SQL

You can call PL/SQL functions of your own creation from within SQL DML statements. You can also call Java methods wrapped in PL/SQL from within SQL. However, these methods must conform to the following purity rules:

- If you call a method from a SELECT statement or a parallelized INSERT, UPDATE, MERGE, or DELETE statement, the method is not allowed to modify any database tables.

- If you call a method from an INSERT, UPDATE, MERGE, or DELETE statement, the method cannot query or modify any database tables modified by that statement.
- If you call a method from a SELECT, INSERT, UPDATE, MERGE, or DELETE statement, the method cannot execute SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM). The method also cannot execute DDL statements because they automatically perform a commit in your session. Note that these restrictions are waived if the method is executed from within an autonomous transaction PL/SQL block.

The objective of these restrictions is to control side effects that might disrupt your SQL statements. If you try to execute a SQL statement that calls a method violating any of these rules, you will receive a runtime error when the SQL statement is parsed.

Exception Handling with Java

On the one hand, the Java exception-handling architecture is very similar to that of PL/SQL. In Java-speak, you throw an exception and then catch it. In PL/SQL-speak, you raise an exception and then handle it.

On the other hand, exception handling in Java is much more robust. Java offers a foundation class called Exception. All exceptions are objects based on that class, or on classes derived from (extending) that class. You can pass exceptions as parameters and manipulate them pretty much as you would objects of any other class.

When a Java stored method executes a SQL statement and an exception is thrown, that exception is an object from a subclass of java.sql.SQLException. That subclass contains two methods that return the Oracle error code and error message: getErrorCode( ) and getMessage( ).

If a Java stored procedure called from SQL or PL/SQL throws an exception that is *not* caught by the JVM, the caller gets an exception thrown from a Java error message. This is how all uncaught exceptions (including non-SQL exceptions) are reported. Let's take a look at the different ways of handling errors and the resulting output.

Suppose that I create a class that relies on JDBC to drop objects in the database (this is drawn from an example in the Oracle documentation):

```
/* File on web: DropAny.java */
import java.sql.*;
import java.io.*;
import oracle.jdbc.driver.*;

public class DropAny {
  public static void object (String object_type, String object_name)
  throws SQLException {
    // Connect to Oracle using JDBC driver
    Connection conn = new OracleDriver().defaultConnection();
```

```
        // Build SQL statement
        String sql = "DROP " + object_type + " " + object_name;
        Statement stmt = conn.createStatement();
        try {
          stmt.executeUpdate(sql);
        }
        catch (SQLException e) {
          System.err.println(e.getMessage());
        }
        finally {
          stmt.close();
        }
      }
    }
```

This example traps and displays any SQLException using the highlighted code. The "finally" clause ensures that the close( ) method executes whether the exception is raised or not, in order to get the statement's open cursor handled properly.

While it doesn't really make any sense to rely on JDBC to drop objects because this can be done much more easily in native dynamic SQL, building it in Java makes the functionality available to other Java programs without calling PL/SQL.

I load the class into the database using *loadjava* and then wrap this class inside a PL/SQL procedure as follows:

```
PROCEDURE dropany (
   tp IN VARCHAR2,
   nm IN VARCHAR2
   )
AS LANGUAGE JAVA
   NAME 'DropAny.object (
            java.lang.String,
            java.lang.String)';
```

When I attempt to drop a nonexistent object, I will see one of two outcomes:

```
SQL> CONNECT scott/tiger
Connected.

SQL> SET SERVEROUTPUT ON
SQL> BEGIN dropany ('TABLE', 'blip'); END;
/
PL/SQL procedure successfully completed.

SQL> CALL DBMS_JAVA.SET_OUTPUT (1000000);

Call completed.

SQL> BEGIN dropany ('TABLE', 'blip'); END;
/
```

```
ORA-00942: table or view does not exist
```

What you see in these examples is a reminder that output from System.err.println will *not* appear on your screen until you explicitly enable it with a call to DBMS_JAVA.SET_OUTPUT. In either case, however, no exception was raised back to the calling block because it was caught inside Java. After the second call to dropany, you can see that the error message supplied through the getMessage( ) method is taken directly from Oracle.

If I comment out the exception handler in the DropAny.object( ) method, I will get something like this (assuming SERVEROUTPUT is enabled, as well as Java output):

```
SQL > BEGIN
  2    dropany('TABLE', 'blip');
  3  EXCEPTION
  4     WHEN OTHERS
  5     THEN
  6        DBMS_OUTPUT.PUT_LINE(SQLCODE);
  7        DBMS_OUTPUT.PUT_LINE(SQLERRM);
  8  END;
  9  /
oracle.jdbc.driver.OracleSQLException: ORA-00942: table or view does not exist
  at oracle.jdbc.driver.T2SConnection.check_error(T2SConnection.java:120)
  at oracle.jdbc.driver.T2SStatement.check_error(T2SStatement.java:57)
  at oracle.jdbc.driver.T2SStatement.execute_for_rows(T2SStatement.java:486)
  at oracle.jdbc.driver.OracleStatement.doExecute
WithTimeout(OracleStatement.java:1148)
  at oracle.jdbc.driver.OracleStatement.executeUpdate(OracleStatement.java:1705)
  at DropAny.object(DropAny:14)

-29532
ORA-29532: Java call terminated by uncaught Java exception: java.sql.SQLException:
ORA-00942: table or view does not exist
```

This takes a little explaining. Everything between:

```
java.sql.SQLException: ORA-00942: table or view does not exist
```

and:

```
-29532
```

represents an error stack dump generated by Java and sent to standard output, regardless of how you handle the error in PL/SQL. In other words, even if my exception section looked like this:

```
EXCEPTION WHEN OTHERS THEN NULL;
```

I would still get all that output on the screen, and then processing in the outer block (if any) would continue. The last three lines of output displayed are generated by my calls to DBMS_OUTPUT.PUT_LINE.

Notice that the Oracle error is not ORA-00942, but instead is ORA-29532, a generic Java error. This is a problem. If you trap the error, how can you discover what the real error is? Looks like it's time for Write-A-Utility Man!

It appears to me that the error returned by SQLERRM is of this form:

```
ORA-29532: Java call ...: java.sql.SQLException: ORA-NNNNN ...
```

So I can scan for the presence of java.sql.SQLException and then SUBSTR from there. The book's web site contains a program in the *getErrorInfo.sp* file that returns the error code and message for the current error, building in the smarts to compensate for the Java error message format.

The main focus in the following sections is an expansion of the JDelete class into the JFile class, which will provide significant new file-related features in PL/SQL. Following that, we'll explore how to write Java classes and PL/SQL programs around them to manipulate Oracle objects.

Extending File I/O Capabilities

Oracle's UTL_FILE package (described in Chapter 22) is notable more for what it is missing than for what it contains. With UTL_FILE, you can read and write the contents of files sequentially. That's it. At least before Oracle9*i* Database Release 2, you can't delete files, change privileges, copy a file, obtain the contents of a directory, set a path, etc., etc. Java to the rescue! Java offers lots of different classes to manipulate files. You've already met the File class and seen how easy it is to add the "delete a file" capability to PL/SQL.

I will now take my lessons learned from JDelete and the rest of this chapter and create a new class called JFile, which will allow PL/SQL developers to answer the questions and take the actions listed here:

- Can I read from a file? Write to a file? Does a file exist? Is the named item a file or a directory?
- What is the number of bytes in a file? What is the parent directory of a file?
- What are the names of all the files in a directory that match a specified filter?
- How can I make a directory? Rename a file? Change the extension of a file?

I won't explain all the methods in the JFile class and its corresponding package; there is a *lot* of repetition, and most of the Java methods look just like the delete( ) function I built at the beginning of the chapter. I will instead focus on the unique issues addressed in different areas of the class and package. You can find the full definition of the code in the following files on the book's web site:

JFile.java
: A Java class that draws together various pieces of information about operating system files and offers it through an API accessible from PL/SQL.

xfile.pkg
The PL/SQL package that wraps the JFile class. Stands for "eXtra stuff for FILEs."

Oracle9*i* Database Release 2 introduced an enhanced version of the UTL_FILE package that, among other things, allows you to delete a file using the UTL_FILE.FREMOVE procedure. It also supports file copying (FCOPY) and file renaming (FRENAME).

Polishing up the delete method

Before moving on to new and exciting stuff, we should make sure that what we've done so far is optimal. The way I defined the JDelete.delete( ) method and the delete_file function is far from ideal. Here's the method code I showed you earlier:

```
public static int delete (String fileName) {
   File myFile = new File (fileName);
   boolean retval = myFile.delete();
   if (retval) return 1; else return 0;
   }
```

And the associated PL/SQL:

```
FUNCTION fDelete (
   file IN VARCHAR2) RETURN NUMBER
AS LANGUAGE JAVA
   NAME 'JDelete.delete (java.lang.String)
            return int';
```

So what's the problem? The problem is that I have been forced to use clumsy, numeric representations for TRUE/FALSE values. As a result, I must write code like this:

```
IF fdelete ('c:\temp\temp.sql') = 1 THEN ...
```

and that is very ugly, hardcoded software. Not only that, but the person writing the PL/SQL code would be required to know about the values for TRUE and FALSE embedded within a Java class.

I would much rather define a delete_file function with this header:

```
 FUNCTION fDelete (
   file IN VARCHAR2) RETURN BOOLEAN;
```

So let's see what it would take to present that clean, easy-to-use API to users of the xfile package.

First, I will rename the JDelete class to JFile to reflect its growing scope. Then, I will add methods that encapsulate the TRUE/FALSE values its other methods will return —and call those inside the delete( ) method. Here is the result:

```
/* File on web: JFile.java */
import java.io.File;

public class JFile {
   public static int tVal () { return 1; };
   public static int fVal () { return 0; };

   public static int delete (String fileName) {
      File myFile = new File (fileName);
      boolean retval = myFile.delete();
      if (retval) return tVal();
         else return fVal();
      }
}
```

That takes care of the Java side of things; now it's time to shift attention to my PL/SQL package. Here's the first pass at the specification of xfile:

```
/* File on web: xfile.pkg */
PACKAGE xfile
IS
   FUNCTION delete (file IN VARCHAR2)
      RETURN BOOLEAN;
END xfile;
```

So now we have the Boolean function specified. But how do we implement it? I have two design objectives:

- Hide the fact that I am relying on numeric values to pass back TRUE or FALSE.
- Avoid hardcoding the 1 and 0 values in the package.

To achieve these objectives, I will define two global variables in my package to hold the numeric values:

```
/* File on web: xfile.pkg */
PACKAGE BODY xfile
IS
   g_true INTEGER;
   g_false INTEGER;
```

And way down at the end of the package body, I will create an initialization section that calls these programs to initialize my globals. By taking this step in the initialization section, I avoid unnecessary calls (and overhead) to Java methods:

```
BEGIN
   g_true := tval;
   g_false := fval;
END xfile;
```

Back up in the declaration section of the package body, I will define two private functions whose only purpose is to give me access in my PL/SQL code to the JFile methods that have encapsulated the 1 and 0:

```
FUNCTION tval RETURN NUMBER
AS LANGUAGE JAVA
   NAME 'JFile.tVal () return int';

FUNCTION fval RETURN NUMBER
```

```
AS LANGUAGE JAVA
   NAME 'JFile.fVal () return int';
```

I have now succeeded in softcoding the TRUE/FALSE values in the JFile package. To enable the use of a true Boolean function in the package specification, I create a private "internal delete" function that is a wrapper for the JFile.delete( ) method. It returns a number:

```
FUNCTION Idelete (file IN VARCHAR2) RETURN NUMBER
AS LANGUAGE JAVA
   NAME 'JFile.delete (java.lang.String) return int';
```

Finally, my public delete function can now call Idelete and convert the integer value to a Boolean by checking against the global variable:

```
FUNCTION delete (file IN VARCHAR2) RETURN BOOLEAN
AS
BEGIN
   RETURN Idelete (file) = g_true;
EXCEPTION
   WHEN OTHERS
   THEN
      RETURN FALSE;
END;
```

And that is how you convert a Java Boolean to a PL/SQL Boolean. You will see this method employed again and again in the xfile package body.

Obtaining directory contents

One of my favorite features of JFile is its ability to return a list of files found in a directory. It accomplishes this feat by calling the File.list( ) method; if the string you used to construct a new File object is the name of a directory, it returns a array of String filenames found in that directory. Let's see how I can make this information available as a collection in PL/SQL.

First, I create a collection type with which to declare these collections:

```
CREATE OR REPLACE TYPE dirlist_t AS TABLE OF VARCHAR2(512);
```

I next create a method called dirlist, which returns an oracle.sql.ARRAY:

```
/* File on web: JFile.java */
import java.io.File;
import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.*;

public class JFile {
...
   public static oracle.sql.ARRAY dirlist (String dir)
      throws java.sql.SQLException
   {
      Connection conn = new OracleDriver().defaultConnection();
      ArrayDescriptor arraydesc =
         ArrayDescriptor.createDescriptor ("DIRLIST_T", conn);

      File myDir = new File (dir);
      String[] filesList = myDir.list();

      ARRAY dirArray = new ARRAY(arraydesc, conn, filesList);
      return dirArray;
   }
...
```

This method first retrieves a "descriptor" of the user-defined type dirlist_t so that we can instantiate a corresponding object. After calling Java's File.list() method, it copies the resulting list of files into the ARRAY object in the invocation of the constructor.

Over on the PL/SQL side of the world, I then create a wrapper that calls this method:

```
FUNCTION dirlist (dir IN VARCHAR2)
   RETURN dirlist_t
AS
   LANGUAGE JAVA
   NAME 'myFile.dirlist(java.lang.String) return oracle.sql.ARRAY';
```

And here is a simple example of how it might be invoked:

```
DECLARE
   tempdir dirlist_t;
BEGIN
   tempdir := dirlist('C:\temp');
   FOR indx IN 1..tempdir.COUNT
   LOOP
      DBMS_OUTPUT.PUT_LINE_(tempdir(indx));
   END LOOP;
END;
```

You will find in the xfile package additional programs to do the following: retrieve filenames as a list rather than an array, limit files retrieved by designated wildcard filters, and change the extension of specified files. You will also find all of the entry points of the UTL_FILE package, such as FOPEN and PUT_LINE. I add those so that you can avoid the use of UTL_FILE for anything but declarations of file handles as UTL_FILE.FILE_TYPE and references to the exceptions declared in UTL_FILE.

Other Examples

On the book's web site there are still more interesting examples of using Java to extend the capabilities of PL/SQL or perform more complex datatype mapping:

utlzip.sql

Courtesy of reviewer Vadim Loevski, this Java class and corresponding package make zip/compression functionality available in PL/SQL. They also use the CREATE OR REPLACE JAVA statement to load a class directly into the database without relying on the *loadjava* command. Here is the header of the Java class creation statement:

```
/* File on web: utlzip.sql */
JAVA SOURCE NAMED "UTLZip" AS
import java.util.zip.*;
import java.io.*;
public class utlzip
{ public static void compressfile(string infilename, string outfilename)
...
}
```

And here is the "cover" for the Java method:

```
PACKAGE utlzip
IS
   PROCEDURE compressfile (p_in_file IN VARCHAR2, p_out_file IN VARCHAR2)
   AS
   LANGUAGE JAVA
      NAME 'UTLZip.compressFile(java.lang.String,
                java.lang.String)';
END;
```

If you are running Oracle Database 10*g* or later, you may not find this quite as useful, because you can just use Oracle's built-in package UTL_COMPRESS instead.

DeleteFile.java and deletefile.sql

Courtesy of reviewer Alex Romankeuich, this Java class and corresponding PL/SQL code demonstrate how to pass a collection (nested table or VARRAY) into an array in Java. The specific functionality implements the deletion of all files in the specified directory that have been modified since a certain date. To create the PL/SQL side of the equation, I first create a nested table of objects, and then pass that collection to Java through the use of the oracle.sql.ARRAY class:

```
CREATE TYPE file_details AS OBJECT (
   dirname      VARCHAR2 (30),
   deletedate   DATE)
/
CREATE TYPE file_table AS TABLE OF file_details;
/
CREATE OR REPLACE PACKAGE delete_files
IS
   FUNCTION fdelete (tbl IN file_table) RETURN NUMBER
   AS
   LANGUAGE JAVA
      NAME 'DeleteFile.delete(oracle.sql.ARRAY) return int';
END delete_files;
```

And here are the initial lines of the Java method. Note that Alex extracts the result set from the array structure and then iterates through that result set. See the *DeleteFile.java* script for the full implementation and extensive comments.

```
/* File on web: DeleteFile.java and deletfile.sql */
public class DeleteFile {
  public static int delete(oracle.sql.ARRAY tbl) throws SQLException {
    try {
```

```
// Retrieve the contents of the table/varray as a result set
ResultSet rs = tbl.getResultSet();

for (int ndx = 0; ndx < tbl.length(); ndx++) {
  rs.next();

  // Retrieve the array index and array element.
  int aryndx = (int)rs.getInt(1);
  STRUCT obj = (STRUCT)rs.getObject(2);
```

utlcmd.sql
: Courtesy of reviewer Vadim Loevski, this Java class and corresponding package make it dangerously easy to execute any operating system command from within PL/SQL. Use with caution.

CHAPTER 28

External Procedures

In the early days of PL/SQL, it was common to hear the question "Can I call *whatever* from within Oracle?" Typically, *whatever* had something to do with sending email, running operating-system commands, or using some non-PL/SQL feature. Although email has pretty much been a nonissue since Oracle began shipping the built-in UTL_SMTP and UTL_MAIL packages, there are by now quite a handful of alternatives to calling "whatever." Here are the most common approaches:

- Write the program as a Java stored procedure and call the Java from PL/SQL.
- Use a database table or queue as a place to store the requests, and create a separate process to read and respond to those requests.
- Expose the program as a web service.
- Use a database pipe and write a daemon that responds to requests on the pipe.
- Write the program in C and call it as an external procedure.

Java may work well, and it can be fast enough for many applications. Queuing is a very interesting technology, but even if you are simply using plain tables, this approach requires two Oracle database sessions: one to write to the queue and one to read from it. Moreover, two sessions means two different transaction spaces, and that might be a problem for your application. Database pipe-based approaches also have the two-session problem, not to mention the challenge of packing and unpacking the contents of the pipe. In addition, handling many simultaneous requests using any of these approaches might require you to create your own listener and process-dispatching system.

Those are all reasons to consider the final option. *External procedures* allow PL/SQL to do almost anything that any other language can do, and can remedy the shortcomings of the other approaches just mentioned. But ... just how do external procedures work? Are they secure? How can I build my own? What are their advantages and disadvantages? This chapter addresses these questions and provides examples of commonly used features of external procedures.

By the way, examples in this chapter make use of the GNU C compiler, which, like Oracle, runs practically everywhere.[*] I love GCC, but it won't work in every situation; you might need to use the compiler that is "native" on your hardware.

Introduction to External Procedures

To call an external program from inside Oracle, the program must run as a shared library. You probably know this type of program as a DLL file (dynamically linked library) on Microsoft operating systems; on Solaris, AIX, and Linux, you'll usually see shared libraries with a *.so* (shared object) file extension, or *.sl* (shared library) on HP-UX. In theory, you can write the external routine in any language you wish, but your compiler and linker will need to generate the appropriate shared library format that is callable from C. You "publish" the external program by writing a special PL/SQL wrapper, known as a *call specification*. If the external function returns a value, it maps to a PL/SQL function; if the external function returns nothing, it maps to a PL/SQL procedure.

Example: Invoking an Operating System Command

Our first example allows a PL/SQL program to execute any operating system-level command. Eh? I hope your mental security buzzer is going off—that sounds like a really dangerous thing to do, doesn't it? Despite several security hoops you have to jump through to make it work, your database administrator will still object to granting wide permissions to run this code. Just try to suspend your disbelief as we walk through the examples.

The first example consists of a very simple C function, extprocsh( ), which accepts a string and passes it to the `system` function for execution:

```
int extprocshell(char *cmd)
{
   return system(cmd);
}
```

The function returns the result code as provided by `system`, a function normally found in the C runtime library (*libc*) on Unix, or in *msvcrt.dll* on Microsoft platforms.

After saving the source code in a file named *extprocsh.c*, I can use the GNU C compiler to generate a shared library. On my 64-bit Solaris machine running GCC 3.4.2 and Oracle Database 10g Release 2, I used the following compiler command. Note that GCC options vary on different Unix/Linux distributions.

```
gcc -m64 extprocsh.c -fPIC -G -o extprocsh.so
```

[*] GNU C is free to use and is found here: *http://gcc.gnu.org*. There are at least two versions for Microsoft Windows; this chapter uses the one from *http://www.mingw.org*.

Similarly, on Microsoft Windows XP Pro running GCC 3.2.3 from Minimal GNU for Windows (MinGW), also with Oracle Database 10g Release 2, this works:

```
c:\MinGW\bin\gcc extprocsh.c -shared -o extprocsh.dll
```

These commands generate a shared library file, *extprocsh.so* or *extprocsh.dll*. Now I need to put the library file somewhere that Oracle can find it. Depending on your Oracle version, that may be easier said than done! Table 28-1 gives you a clue as to where to put the files.

Table 28-1. Location of shared library files

Version	Default "allowed" location	Means of specifying nondefault location
Oracle8 Database	Anywhere readable by the oracle process	Not applicable
Oracle8 *i* Database		
Oracle9*i* Database Release 1		
Oracle9*i* Release 2 and later	*$ORACLE_HOME/lib* and/or *$ORACLE_HOME/bin* (varies by Oracle version and platform; *lib* is typical for Unix and *bin* for Microsoft Windows)	Edit listener configuration file and supply path value(s) for ENVS="EXTPROC_DLLS..." property (see the section "“The Oracle Net Configuration” on page 1176“)

After copying the file and/or making adjustments to the listener, I also need to define a "library" inside Oracle to point to the DLL:

```
CREATE OR REPLACE LIBRARY extprocshell_lib
   AS '/u01/app/oracle/local/lib/extprocsh.so';   -- Unix-Linux

CREATE OR REPLACE LIBRARY extprocshell_lib
   AS 'c:\oracle\local\lib\extprocsh.dll';      -- Microsoft
```

Don't by confused by the term "library" here; it's really just a filename alias that can be used in Oracle's namespace. Also note that performing this step requires Oracle's CREATE LIBRARY privilege, which is one of the security hoops I mentioned earlier.

Now I can create a PL/SQL call specification that uses the newly created library:

```
FUNCTION shell(cmd IN VARCHAR2)
   RETURN PLS_INTEGER
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocshell"
   PARAMETERS (cmd STRING, RETURN INT);
```

That's all there is to it! Assuming that the DBA has set up the system environment to support external procedures (see the section "Specifying the Listener Configuration" on page 1176 later in this chapter), shell( ) is now usable anywhere you can invoke a PL/SQL function—SQL*Plus, Perl, Pro*C, etc. From an application programming perspective, calling an external procedure is indistinguishable from calling a conventional procedure. For example:

```
DECLARE
   result PLS_INTEGER;
BEGIN
   result := shell('cmd');
END;
```

Or even:

```
SQL> SELECT shell('cmd') FROM DUAL;
```

If successful, this will return zero:

```
SHELL('cmd')
-------------
          0
```

Keep in mind that if the operating-system command normally displays output to stdout or stderr, that output will go to the bit bucket unless you modify your program to return it to PL/SQL. You can, subject to OS-level permissions, redirect that output to a file; here is a trivial example of saving a file containing a directory listing:

```
result := shell('ls / > /tmp/extproc.out'));                 -- Unix-Linux
result := shell('cmd /c "dir c:\ > c:\temp\extproc.out"'));  -- Microsoft
```

These operating-system commands will execute with the same privileges as the Oracle Net listener that spawns the *extproc* process. Hmmm, I'll bet your DBA or security guy will want to change *that*. Read on if you want to help.

Architecture of External Procedures

What happens under the covers when you invoke an external procedure? Let's first consider a case such as the example illustrated in the previous section, which uses the default external procedure "agent."

When the PL/SQL runtime engine learns from the compiled code that the program has been implemented externally, it looks for a TNS service named EXTPROC_CONNECTION_DATA, which must be known to the server via some Oracle Net naming method such as the *tnsnames.ora* file. As shown in Figure 28-1, the Oracle Net listener responds to the request by spawning a session-specific process called *extproc*, to which it passes the path to the DLL file along with the function name and any arguments. It is *extproc* that dynamically loads your shared library, sends needed arguments, receives its output, and transmits these results back to the caller. In this arrangement, only one *extproc* process runs for a given Oracle session; it launches with the first external procedure call and terminates when the session disconnects. For each distinct external procedure you call, this *extproc* process loads the associated shared library (if it hasn't already been loaded).

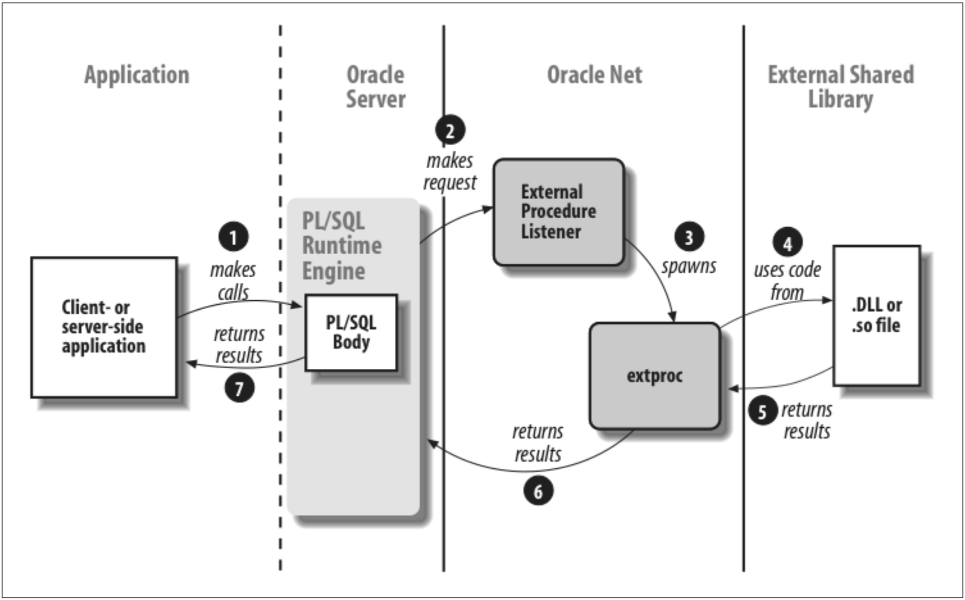


Figure 28-1. Invoking an external procedure that uses the default agent

Oracle has provided a number of features to help make external procedures usable and efficient:

Shared DLL
: The external C program must be in a shared dynamically linked library rather than in a statically linked module. Although deferring linking until runtime incurs some overhead, there should be memory savings when more than one session uses a shared library; the operating system allows some of the memory pages of the library to be shared by more than one process. Another benefit of using dynamically linked modules is that they can be created and updated more easily than statically linked programs. In addition, there can be many subprograms in a shared library (hence the term "library"). This mitigates the performance overhead by allowing you to load fewer files dynamically.

Separate memory space
: Oracle external procedures run in a separate memory space from the main database kernel processes. If the external procedure crashes, it won't step on kernel memory; the *extproc* process simply returns an error to the PL/SQL engine, which in turn reports it to the application. Writing an external procedure to crash the Oracle server is possible, but it's no easier than doing so from a non-external procedure program.

Full transaction support
: External procedures provide full transaction support; that is, they can participate fully in the current transaction. By accepting "context" information from PL/SQL,

the procedure can call back to the database to fetch data, make SQL or PL/SQL calls, and raise exceptions. Using these features requires some low-level Oracle Call Interface (OCI) programming ... but at least it's possible!

Multithreading (Oracle Database 10g and later)
Up through Oracle9*i* Database, each Oracle session that called an external procedure required a companion *extproc* process. For large numbers of users, the added overhead could be significant. Starting with Oracle Database 10*g*, though, the DBA can configure a multithreaded "agent" that services each request in a thread rather than a separate process. You will need to ensure that your C program is thread-safe if you go this route. See the later section "Setting Up Multithreaded Mode" on page 1179 for more information about using this feature.

Despite their many features and benefits, external procedures are not a perfect match for every application: Oracle's architecture requires an unavoidable amount of inter-process communication. This is the tradeoff required for the safety of separating the external procedure's memory space from that of the database server.

The Oracle Net Configuration

Let's take a look at how you would set up a simple configuration that will support external procedures while closing up some of the glaring security gaps.

Specifying the Listener Configuration

It is the Oracle Net communications layer that provides the conduit between PL/SQL and the shared libraries. Although default installations of Oracle8*i* Database and later generally provide some support for external procedures, you probably don't want to use the out-of-the-box configuration until Oracle has made some significant security enhancements.

At the time we were writing the third edition of this book, Oracle was suffering a bit of a black eye from a security vulnerability arising from the external procedures feature. Specifically, a remote attacker could connect via the Oracle Net TCP/IP port (usually 1521) and run *extproc* with no authentication. Although Oracle closed up that particular vulnerability, the conventional wisdom of securing Oracle includes that shown in the following note.

Keep Oracle listeners behind a firewall; never expose a listener port to the Internet or to any other untrusted network.

Getting the listener set up properly involves modifying the *tnsnames.ora* file and the *listener.ora* file (either by hand or by using the Oracle Net Manager frontend). Here, for example, is a simple *listener.ora* file that sets up an external procedure listener that is separate from the database listener:

```
### regular listener (to connect to the database)

LISTENER =
   (ADDRESS = (PROTOCOL = TCP)(HOST = hostname)(PORT = 1521))

SID_LIST_LISTENER =
   (SID_DESC =
      (GLOBAL_DBNAME = global_name)
      (ORACLE_HOME = oracle_home_directory)
      (SID_NAME = SID)
   )

#### external procedure listener

EXTPROC_LISTENER =
   (ADDRESS = (PROTOCOL = IPC)(KEY = extprocKey))

SID_LIST_EXTPROC_LISTENER =
   (SID_DESC =
      (SID_NAME = extprocSID)
      (ORACLE_HOME = oracle_home_directory)
      (ENVS="EXTPROC_DLLS=shared_object_file_list,other_envt_vars")
      (PROGRAM = extproc)
   )
```

where:

hostname, global_name
: *hostname* is the name or IP address of this machine; *global_name* is the fully qualified name of the database. In the example, these parameters apply to the database listener only, not to the external procedure listener.

extprocKey
: A short identifier used by Oracle Net to distinguish this listener from other potential IPC listeners. Its actual name is arbitrary because your programs will never see it. Oracle uses EXTPROC0 or EXTPROC1 as the default name for the first Oracle Net installation on a given machine. This identifier must be the same in the address list of *listener.ora* and in the *tnsnames.ora* file.

oracle_home_directory
: The full pathname to your ORACLE_HOME directory, such as */u01/app/oracle/oracle/product/10.2.0/db_1* on Unix or *C:\oracle\product\10.2.0\db_1* on Microsoft Windows. Notice that there are no quotation marks around the directory name and no trailing slash.

extprocSID
: An arbitrary unique identifier for the external procedure listener. In the default installation, Oracle uses the value PLSExtProc.

ENVS="EXTPROC_DLLS=shared_object_file_list"

Available in Oracle9*i* Database Release 2 and later. The ENVS clause sets up environment variables for the listener. The list is colon-delimited, and each element must be the fully qualified path name to the shared object file.

There are some special keywords you can use in the list. For no security at all, you can use the ANY keyword, which lets you use any shared library that is visible to the operating system user running the external procedure listener. In other words:

```
ENVS="EXTPROC_DLLS=ANY"
```

For maximum security, use the ONLY keyword to limit execution of those shared libraries given by the colon-delimited list. Here is an example from my Solaris machine that shows what this might look like:

```
(ENVS="EXTPROC_DLLS=ONLY:/u01/app/oracle/local/lib/extprocsh.so:/u01/app/oracle/
local/lib/RawdataToPrinter.so")
```

And here is an entry from my laptop machine, which runs a Microsoft Windows operating system:

```
(ENVS="EXTPROC_DLLS=ONLY:c:\oracle\admin\local\lib\extprocsh.dll:c:\oracle\admin\
local\lib\RawDataToPrinter.dll")
```

Here, the colon symbol has two different meanings; as the list delimiter or as the drive letter separator. Also note that although I've shown only two library files, you can include as many as you like.

If you omit the ANY and ONLY keywords but still provide a list of files, both the default directories and the explicitly named files are available.

other_envt_vars

You can set values for environment variables needed by shared libraries by adding them to the ENVS setting of the external procedure listener. A commonly needed value on Unix is LD_LIBRARY_PATH:

```
(ENVS="EXTPROC_DLLS=shared_object_file_list,LD_LIBRARY_PATH=/usr/local/lib")
```

Use commas to separate the list of files and each environment variable.

Security Characteristics of the Configuration

The configuration established here accomplishes two important security objectives:

- It allows the system administrator to run the external procedure listener as a user account with limited privileges. By default, the listener would run as the account that runs the Oracle server.
- It limits the external procedure listener to accept only IPC connections from the local machine, as opposed to TCP/IP connections from anywhere.

But we're not quite done. The *tnsnames.ora* file for the database in which the callout originates will need an entry like the following:

```
EXTPROC_CONNECTION_DATA =
   (DESCRIPTION =
      (ADDRESS = (PROTOCOL = IPC)(KEY = extprocKey))
      (CONNECT_DATA = (SID = extprocSID) (PRESENTATION = RO))
   )
```

You'll recognize most of these settings from the earlier listener configuration. Note that the values you used in the listener for *extprocKey* and *extprocSID* must match their respective values here. The optional PRESENTATION setting is intended to improve performance a bit; it tells the server, which might be listening for different protocols, to assume that the client wants to communicate using the protocol known as Remote-Ops (hence the RO).

You'll want to be careful about what privileges the supplemental listener account has, especially regarding its rights to modify files owned by the operating system or by the *oracle* account. Also, by setting the TNS_ADMIN environment variable on Unix (or in the registry of a Microsoft operating system), you can relocate the external procedure listener's *listener.ora* and *sqlnet.ora* files to a separate directory. This may be another aspect of an overall approach to security.

Setting up these configuration files and creating supplemental OS-level user accounts may seem rather distant from day-to-day PL/SQL programming, but these days, security is everybody's business!

Oracle professionals should keep up with Oracle's security alerts page at *http://otn.oracle.com/deploy/security/alerts.htm*. The external procedures problem I mentioned back in the section "Specifying the Listener Configuration" on page 1176 first appeared as alert number 29, but every Oracle shop should regularly review the entire list of issues to discover what workarounds or patches to employ.

Setting Up Multithreaded Mode

Oracle Database 10*g* introduced a way for multiple sessions to share a single external procedure process. Although this feature takes a bit of effort to set up, it could pay off when you have many users running external procedures. Here are the minimum steps required for your DBA to turn on multithreaded mode:

1. Shut down the external procedure listener. If you have configured a separate listener for it as recommended above, this step is simply:

   ```
   OS> lsnrctl stop extproc_listener
   ```

2. Edit *listener.ora*: first, change your *extprocKey* (which by default would be EXTPROC0 or EXTPROC1) to PNPKEY; second, to eliminate the possibility of any dedicated listeners, delete the entire SID_LIST_EXTPROC_LISTENER section.
3. Edit *tnsnames.ora*, changing your *extprocKey* to be PNPKEY.

4. Restart the external procedure listener; for example:

   ```
   OS> lsnrctl start extproc_listener
   ```

5. At the operating system command prompt, be sure you have set a value for the AGTCTL_ADMIN environment variable. The value should consist of a fully qualified directory path; this tells *agtctl* where to store its settings. (If you don't set AGTCTL_ADMIN, but do have TNS_ADMIN set, the latter will be used instead.)
6. If you need to send any environment variables to the agent such as EXTPROC_DLLS or LD_LIBRARY_PATH, set these in the current operating system session. Here are some examples (if using the *bash* shell or equivalent):

   ```
   OS> export EXTPROC_DLLS=ANY
   OS> export LD_LIBRARY_PATH=/lib:/usr/local/lib/sparcv9
   ```

7. Assuming that you are still using the external procedure's default listener "SID," that is, PLSExtProc, run the following:

   ```
   OS> agtctl startup extproc PLSExtProc
   ```

To see if it's working, you can use the "lsnrctl services" command:

```
OS> lsnrctl services extproc_listener
...
Connecting to (ADDRESS=(PROTOCOL=IPC)(KEY=PNPKEY))
Services Summary...
Service "PLSExtProc" has 1 instance(s).
  Instance "PLSExtProc", status READY, has 1 handler(s) for this service...
    Handler(s):
      "ORACLE SERVER" established:0 refused:0 current:0 max:5 state:ready
         PLSExtProc
         (ADDRESS=(PROTOCOL=ipc)(KEY=#5746.1.4))
```

This output is what we hoped to see; the agent is listed in state "ready," and is not labeled as dedicated. This command also shows stats on the number of sessions; in the output above, everything is "0" except the maximum number of sessions, which defaults to 5.

Internally, a multithreaded agent uses its own listener/dispatcher/worker bee arrangement, allowing each session request to get handed off to its own thread of execution. You can control the numbers of tasks using "agtctl set" commands. For example, to modify the maximum number of sessions, first shut down the agent:

```
OS> agtctl shutdown PLSExtProc
```

Then set max_sessions:

```
OS> agtctl set max_sessions n PLSExtProc
```

Where *n* is the maximum number of Oracle sessions that can connect to the agent simultaneously.

Finally, restart the agent:

```
OS> agtctl startup extproc PLSExtProc
```

When tuning your setup, there are several parameter settings to be aware of:

Parameter	Description	Default
max_dispatchers	Maximum number of dispatcher threads, which hand off requests to the task threads	1
max_task_threads	Maximum number of "worker bee" threads	2
max_sessions	Maximum number of Oracle sessions that can be serviced by the multithreaded *extproc* process	5
listener_address	Addresses with which the multithreaded process "registers" with an already-running listener	`(ADDRESS_LIST= (ADDRESS= (PROTOCOL=IPC) (KEY=PNPKEY)) (ADDRESS= (PROTOCOL=IPC) (KEY=`*`listenerSID`*`)) (ADDRESS= (PROTOCOL=TCP) (HOST=127.0.0.1) (PORT=1521)))`

By the way, while testing this feature, I discovered that whenever I bounced the listener, afterwards I needed to bounce the agent as well. Fortunately, the *agtctl* utility kindly remembers any parameter adjustments you have made from the default values.

Some experimentation may be needed to optimize the agent-specific parameters against the number of agent processes. While I have not experimented enough with multi-threaded agents to offer any rules of thumb, let's at least take a look at the changes required to use two multithreaded agents. Follow the steps given earlier, but this time, in Step 7, you will start two agents with unique names:

```
OS> agtctl startup extproc PLSExtProc_001
...
OS> agtctl startup extproc PLSExtProc_002
```

You must also modify *tnsnames.ora* (Step 3 above) to be aware of these new agent names; because you probably want Oracle Net to "load balance" across the agents, edit the EXTPROC_CONNECTION_DATA section of *tnsnames.ora* to be:

```
EXTPROC_CONNECTION_DATA =
   (DESCRIPTION_LIST =
      (LOAD_BALANCE = TRUE)
      (DESCRIPTION =
         (ADDRESS = (PROTOCOL = IPC)(KEY = PNPKEY))
         (CONNECT_DATA = (SID = PLSExtProc_001)(PRESENTATION = RO))
      )
      (DESCRIPTION =
         (ADDRESS = (PROTOCOL = ipc)(key = PNPKEY))
         (CONNECT_DATA = (SID = PLSExtProc_002)(PRESENTATION = RO))
      )
   )
```

You need to add the DESCRIPTION_LIST parameter and include one description section for each of the agents.

With this new configuration, each time Oracle Net receives a request from PL/SQL to connect to an external procedure, Oracle Net will randomly connect to one of the agents listed; if the connection fails (for example, because it already has the maximum number of sessions connected), Oracle Net will try the other agent, until it either makes a successful connection or fails to connect to any of them. Such a failure will result in *ORA-28575: unable to open RPC connection to external procedure agent*.

You can read more about multithreaded mode and *agtctl* in Oracle's *Application Development Guide—Fundamentals* and the *Heterogenous Connectivity Administrator's Guide*. Oracle Net's load balancing features are described in the *Net Services Administrator's Guide* and the *Net Services Reference*.

One final point: when using multithreaded agents, the C program that implements an external procedure must be *thread-safe*, and writing such a beasty is not necessarily trivial. See the notes at the end of this chapter for a few of the caveats.

Creating an Oracle Library

The SQL statement CREATE LIBRARY defines an alias in the Oracle data dictionary for the external shared library file, allowing the PL/SQL runtime engine to find the library when it is called. The only users who can create libraries are administrators and those to whom they have granted the CREATE LIBRARY or CREATE ANY LIBRARY privilege.

The general syntax for the CREATE LIBRARY command is:

```
CREATE [ OR REPLACE ] LIBRARY library_name
AS
   'path_to_file' [ AGENT 'agent_db_link' ] ;
```

where:

library_name
: A legal PL/SQL identifier. This name will be used in subsequent bodies of external procedures that need to call the shared object (or DLL) file. The library name cannot be the same as a table, top-level PL/SQL object, or anything else in the main namespace.

path_to_file
: The fully qualified pathname to the shared object (or DLL) file, enclosed in single quotes.

  In Oracle9*i* Database, it became possible to use environment variables in *path_to_file*. In particular, if the operating system-level account sets the variable before starting the listener, you can put this variable in the CREATE LIBRARY statement; for example:

```
CREATE LIBRARY extprocshell_lib AS '${ORACLE_HOME}/lib/extprocsh.so'; -- Unix
CREATE LIBRARY extprocshell_lib AS '%{ORACLE_HOME}%\bin\extprocsh.dll'; -- MS
```

This may be a good thing to do for the sake of script portability.

You can also use an environment variable that you supply via EXTPROC_DLLS in the *listener.ora* file, as discussed earlier.

AGENT 'agent_db_link'
: (Optional) Database link to which the library owner has access. You must make sure that there is an entry in *tnsnames.ora* for the service name you specify when creating *agent_db_link*, and that the entry includes an external procedure address and connection data. Using the AGENT clause allows the external procedure to run on a different database server, although it must still be on the same machine. The AGENT clause was introduced in Oracle9*i* Database.

Here are some things to keep in mind when issuing a CREATE LIBRARY statement:

- The statement must be executed by the DBA or by a user who has been granted CREATE LIBRARY or CREATE ANY LIBRARY privileges.
- As with most other database objects, libraries are owned by a specific Oracle user (schema). The owner automatically has execution privileges, and can grant and revoke the EXECUTE privilege on the library to other users.
- Other users who have received EXECUTE privilege on a library can refer to it in their own call specs using *owner.library* syntax, or they can create and use synonyms for the library if desired.
- Oracle doesn't check whether the named shared library file exists when you execute the CREATE LIBRARY statement. Nor will it check when you later create an external procedure declaration for a function in that library. If you have an error in the path, you won't know it until the first time you try to execute the function.

You need to create only a single Oracle library in this fashion for each shared library file you use. There can be any number of callable C functions in the library file and any number of call specifications that refer to the library.

Let's take a closer look at how to write a PL/SQL subprogram that maps the desired routine from the shared library into a PL/SQL-callable form.

Writing the Call Specification

An external procedure can serve as the implementation of any program unit other than an anonymous block. In other words, a call specification can appear in a top-level procedure or function, a packaged procedure or function, or an object method. What's more, you can define the call spec in either the specification or the body of packaged program units (or in either the spec or body of object types). Here are some schematic examples:

```
CREATE FUNCTION name (args) RETURN datatype
AS callspec;
```

You should recognize the form shown here as that of the shell( ) function shown earlier in the chapter. You can also create a procedure:

```
CREATE PROCEDURE name
AS callspec;
```

In this case, the corresponding C function would be typed void.

The next form shows a packaged function that does not need a package body:

```
CREATE PACKAGE pkgname
AS
   FUNCTION name RETURN datatype
   AS callspec;
END;
```

However, when the time comes to modify the package, you would have to recompile the specification. Depending on the change you need to make, you may considerably reduce the recompilation ripple effect by moving the call spec into the package body:

```
CREATE PACKAGE pkgname
AS
   PROCEDURE name;
END;

CREATE PACKAGE BODY pkgname
AS
   PROCEDURE name
   AS callspec;
END;
```

Unpublished or private program units inside packages can also be implemented as external procedures. And finally, using a call spec in an object type method is quite similar to using it in a package; that is, you can put the call spec in the object type specification or in the corresponding type body.

The Call Spec: Overall Syntax

It is the AS LANGUAGE clause[†] that distinguishes the call spec from a regular stored program.

Syntactically, the clause looks like this:

```
AS LANGUAGE C
   LIBRARY library_name
   [ NAME external_function_name ]
   [ WITH CONTEXT ]
   [ AGENT IN (formal_parameter_name) ]
   [ PARAMETERS (external_parameter_map) ] ;
```

[†] Oracle8 Database did not have this clause, offering instead a now-deprecated form, AS EXTERNAL.

where:

AS LANGUAGE C
: Another option here is AS LANGUAGE JAVA, as covered in Chapter 27. There are no other supported languages.

library_name
: Name of the library, as defined in a CREATE LIBRARY statement, which you have privilege to execute, either by owning it or by receiving the privilege.

external_function_name
: Name of the function as defined in the C language library. If the name is lowercase or mixed case, you must put double quotes around it. You can omit this parameter, in which case the name of the external routine must match your PL/SQL module's name (defaults to uppercase).

WITH CONTEXT
: The presence of this clause indicates that you want PL/SQL to pass a "context pointer" to the called program. The called program must be expecting the pointer as a parameter of type *OCIExtProcContext* * (defined in the C header file *ociextp.h*).

  This "context" that you are passing via a pointer is an opaque data structure that contains Oracle session information. The called procedure doesn't need to manipulate the data structure's content directly; instead, the structure simply facilitates other OCI calls that perform various Oracle-specific tasks. These tasks include raising predefined or user-defined exceptions, allocating session-only memory (which gets released as soon as control returns to PL/SQL), and obtaining information about the Oracle user's environment.

AGENT IN (formal_parameter_name)
: This clause is a way of designating a different agent process, similar to the AGENT clause on the library, but deferring the selection of the agent until runtime. The idea is that you pass in the value of the agent as a formal PL/SQL parameter to the call spec; it will supersede the name of the agent given in the library, if any. To learn a little more about the AGENT IN clause, see the section "Nondefault Agents" on page 1196.

PARAMETERS (external_parameter_map)
: This section gives the position and datatypes of parameters exchanged between PL/SQL and C. The *external_parameter_map* is a comma-delimited list of elements that match positionally with the parameters in the C function or that supply additional properties.

Getting the mapping right is potentially the most complex task you face, so the next section spends a bit of time examining the wilderness of details.

Parameter Mapping: The Example Revisited

Consider for a moment the problems of exchanging data between PL/SQL and C. PL/SQL has its own set of datatypes that are only somewhat similar to those you find in C. PL/SQL variables can be NULL and are subject to three-valued truth table logic; C variables have no equivalent concept. Your C library might not know which national language character set you're using to express alphanumeric values. And should your C functions expect a given argument by value or by reference (pointer)?

I'd like to start with an example that builds on the shell program illustrated earlier in the chapter. When we last saw the shell( ) function, it had no protection from being called with a NULL argument instead of a real command. It turns out that calling shell (NULL) results in the runtime error *ORA-01405: fetched column value is NULL*. That may be a perfectly acceptable behavior in some applications, but what if I prefer that the external procedure simply respond to a null input with a null output?

Properly detecting an Oracle NULL in C requires PL/SQL to transmit an additional parameter known as an *indicator variable*. Likewise, for the C program to return an Oracle NULL, it must return a separate indicator parameter back to PL/SQL. While Oracle sets and interprets this value automatically on the PL/SQL side, the C application will need to get and set this value explicitly.

It's probably simplest to illustrate this situation by looking at how the PL/SQL call spec will change:

```
FUNCTION shell(cmd IN VARCHAR2)
   RETURN PLS_INTEGER
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocsh"
   PARAMETERS (cmd STRING, cmd INDICATOR, RETURN INDICATOR, RETURN INT);
```

Although the PL/SQL function's formal parameters can appear anywhere in the PARAMETERS mapping, the items in the mapping must correspond in position and in associated datatype with the parameters in the C function. Any RETURN mapping that you need to provide must be the last item on the list.

You can omit RETURN from the parameter map if you want Oracle to use the default mapping (explained later). This would actually be OK in this case, although the indicator still has to be there:

```
FUNCTION shell(cmd IN VARCHAR2)
   RETURN PLS_INTEGER
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocsh"
   PARAMETERS (cmd STRING, cmd INDICATOR, RETURN INDICATOR);
```

The good news is that even though you've made a number of changes to the call spec compared with the version earlier in the chapter, a program that invokes the shell( ) function sees no change in the number or datatype of its parameters.

Let's turn now to the new version of the C program, which adds two parameters, one for each indicator:

```
 1    #include <ociextp.h>
 2
 3    int extprocsh(char *cmd, short cmdInd, short *retInd)
 4    {
 5       if (cmdInd == OCI_IND_NOTNULL)
 6       {
 7          *retInd = (short)OCI_IND_NOTNULL;
 8          return system(cmd);
 9       } else
10       {
11          *retInd = (short)OCI_IND_NULL;
12          return 0;
13       }
14    }
```

The changes you'll notice are summarized in the following table:

Line(s)	Changes
1	This include file appears in the *%ORACLE_HOME%\oci\include* subdirectory on Microsoft platforms; on Unix-like machines, I've spotted this file in *$ORACLE_HOME/rdbms/demo* (Oracle9*i* Database) and *$ORACLE_HOME/rdbms/public* (Oracle Database 10*g*), although it may be somewhere else on your system.
3	Notice that the command indicator is `short`, but the return indicator is `short *`. That follows the argument-passing convention of using call-by-value for input parameters sent from PL/SQL to C, but call-by-reference for output or return parameters sent from C to PL/SQL.
5, 7	The indicator variable is either OCI_IND_NULL or OCI_IND_NOTNULL; these are special #define values from Oracle's include file. Here, explicit assignments in the code set the return indicator to be one or the other.
11–12	The return indicator takes precedence over the return of 0; the latter is simply ignored.

Here are some simple commands that will compile and link the above program, first on my 64-bit Solaris machine (still using GCC):

```
gcc -m64 extprocsh.c -fPIC -G -I$ORACLE_HOME/rdbms/public -o extprocsh.so
```

And here are the equivalent commands on my Microsoft Windows machine (all on one line):

```
c:\MinGW\bin\gcc -Ic:\oracle\product\10.2.0\db_1\oci\include extprocsh.c
-shared -o extprocsh.dll
```

And now, steel yourself to face the intimidating details of parameter mapping.

Parameter Mapping: The Full Story

As shown in the previous section, when moving data between PL/SQL and C, each PL/SQL datatype maps to an external datatype, identified by a PL/SQL keyword, which in turn maps to an allowed set of C types.

You identify an external datatype in the PARAMETERS clause with a keyword known to PL/SQL. In some cases, the external datatypes have the same name as the C type, but in others they don't. For example, if you pass a PL/SQL variable of type PLS_INTEGER, the corresponding default external type is INT, which maps to an int in C. But Oracle's VARCHAR2 type uses the STRING external datatype, which normally maps to a char * in C.

Table 28-2 lists all the possible datatype conversions supported by Oracle's PL/SQL-to-C interface. Note that the allowable conversions depend on both the datatype and the mode of the PL/SQL formal parameter, as the previous example illustrated. The defaults, if ambiguous, are shown in bold in the table.

Table 28-2. Legal mappings of PL/SQL and C datatypes

		C datatypes for PL/SQL parameters that are:	
Datatype of PL/SQL parameter	**PL/SQL keyword identifying external type**	**N or function return values**	**IN OUT, OUT, or any parameter designated as being passed BY REFERENCE**
Long integer family: BINARY_INTEGER, BOOLEAN, PLS_INTEGER	**INT**, UNSIGNED INT, CHAR, UNSIGNED CHAR, SHORT, UNSIGNED SHORT, LONG, UNSIGNED LONG, SB1, UB1, SB2, UB2, SB4, UB4, SIZE_T	int, unsigned int, char, unsigned char, short, unsigned short, long, unsigned long, sb1, ub1, sb2, ub2, sb4, ub4, size_t	Same list of types as at left, but use a pointer (for example, the int * rather than int)
Short integer family: NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE	Same as above, except default is UNSIGNED INT	Same as above, except default is unsigned int	Same as above, except default is unsigned int *
Character family: VARCHAR2, CHAR, NCHAR, LONG, NVARCHAR2, VARCHAR, CHARACTER, ROWID	**STRING**, OCISTRING	**char ***, OCIString *	**char ***, OCIString *
NUMBER	OCINUMBER	OCINumber *	OCINumber *
DOUBLE PRECISION	DOUBLE	double	double *
FLOAT, REAL	FLOAT	float	float *
RAW, LONG RAW	**RAW**, OCIRAW	**unsigned char ***, OCIRaw *	**unsigned char ***, OCIRaw *
DATE	OCIDATE	OCIDate *	OCIDate *

		C datatypes for PL/SQL parameters that are:	
Datatype of PL/SQL parameter	PL/SQL keyword identifying external type	N or function return values	IN OUT, OUT, or any parameter designated as being passed BY REFERENCE
Timestamp family: TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE	OCIDATETIME	OCIDateTime *	OCIDateTime *
INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH	OCIINTERVAL	OCIInterval *	OCIInterval *
BFILE, BLOB, CLOB, NCLOB	OCILOBLOCATOR	OCILOBLOCATOR *	OCILOBLOCATOR **
Descriptor of user-defined type (collection or object)	TDO	OCIType *	OCIType *
Value of user-defined collection	OCICOLL	OCIColl **, OCIArray **, OCITable **	OCIColl **, OCIArray **, OCITable **
Value of user-defined object	DVOID	dvoid *	dvoid * (use dvoid ** for nonfinal types that are IN OUT or OUT)

In some simple cases where you are passing only numeric arguments and where the defaults are acceptable, you can actually omit the PARAMETERS clause entirely. However, you must use it when you want to pass indicators or other data properties.

Each piece of supplemental information we want to exchange will be passed as a separate parameter, and will appear both in the PARAMETERS clause and in the C language function specification.

More Syntax: The PARAMETERS Clause

The PARAMETERS clause provides a comma-delimited list that may contain five different kinds of elements:

- The name of the parameter followed by the external datatype identifier.
- The keyword RETURN and its associated external datatype identifier.
- A "property" of the PL/SQL parameter or return value, such as a nullness indicator or an integer corresponding to its length.
- The keyword CONTEXT, which is a placeholder for the context pointer.
- The keyword SELF, in the case of an external procedure for an object type member method.

Elements (other than CONTEXT) follow the syntax pattern:

```
{pname | RETURN | SELF} [property] [BY REFERENCE] [external_datatype]
```

If your call spec includes WITH CONTEXT, the corresponding element in the parameter list is simply:

```
CONTEXT
```

By convention, if you have specified WITH CONTEXT, you should make CONTEXT the first argument because that is its default location if the rest of the parameter mappings are defaulted.

Parameter entries have the following meanings:

pname | RETURN | SELF
: The name of the parameter as specified in the formal parameter list of the PL/SQL module, or the keyword RETURN, or the keyword SELF (in the case of a member method in an object type). PL/SQL parameter names do not need to be the same as the names of formal parameters in the C language routine. However, parameters in the PL/SQL parameter list must match one for one, in order, those in the C language specification.

property
: One of the following: INDICATOR, INDICATOR STRUCT, LENGTH, MAXLEN, TDO, CHARSETID, or CHARSETFORM. These are described in the next section.

BY REFERENCE
: Pass the parameter by reference. In other words, the module in the shared library is expecting a pointer to the parameter rather than its value. BY REFERENCE only has meaning for scalar IN parameters that are not strings, such as BINARY_INTEGER, PLS_INTEGER, FLOAT, DOUBLE PRECISION, and REAL. All others (IN OUT and OUT parameters, as well as IN parameters of type STRING) are *always* passed by reference, and the corresponding C prototype must specify a pointer.

external_datatype
: The external datatype keyword from the second column of Table 28-2. If this is omitted, the external datatype will default as indicated in the table.

PARAMETERS Properties

This section describes each possible property you can specify in a PARAMETERS clause.

The INDICATOR property

The INDICATOR property is a flag to denote whether the parameter is null, and has the following characteristics:

Allowed external types
: short (the default), int, long

Allowed PL/SQL types
: All scalars can use an INDICATOR; to pass indicator variables for composite types such as user-defined objects and collections, use the INDICATOR STRUCT property.

Allowed PL/SQL modes
: IN, IN OUT, OUT, RETURN

Call mode
: By value for IN parameters (unless BY REFERENCE specified), and by reference for IN OUT, OUT, and RETURN variables.

You can apply this property to any parameter, in any mode, including RETURNs. If you omit an indicator, PL/SQL is supposed to think that your external routine will always be non-null (but it's not that simple; see the sidebar "Indicating Without Indicators?" on page 1191).

When you send an IN variable and its associated indicator to the external procedure, Oracle sets the indicator's value automatically. However, if your C module is returning a value in a RETURN or OUT parameter and an indicator, your C code must set the indicator value.

For an IN parameter, the indicator parameter in your C function might be:

```
short pIndicatorFoo
```

Or for an IN OUT parameter, the indicator might be:

```
short *pIndicatorFoo
```

In the body of your C function, you should use the #define constants OCI_IND_NOTNULL and OCI_IND_NULL, which will be available in your program if you #include *oci.h*. Oracle defines these as:

```
typedef sb2 OCIInd;
#define OCI_IND_NOTNULL (OCIInd)0          /* not NULL */
#define OCI_IND_NULL (OCIInd)(-1)          /* NULL */
```

Indicating Without Indicators?

What happens if you don't specify an indicator variable for a string and then return an empty C string? We wrote a short test program to find out:

```
void mynull(char *outbuff){
    outbuff[0] = '\0';
}
```

The call spec could look like this:

```
CREATE OR REPLACE PROCEDURE mynull
   (res OUT VARCHAR2)
AS
   LANGUAGE C
   LIBRARY mynulllib
   NAME "mynull";
```

When invoked as an external procedure, PL/SQL does actually interpret this parameter value as a NULL. The reason appears to be that the STRING external type is special; you can also indicate a NULL value to Oracle by passing a string of length 2 where the first byte is \0. (This works only if you omit a LENGTH parameter.)

But you probably shouldn't take this lazy way out; use an indicator instead!

The LENGTH property

The LENGTH property is an integer indicating the number of characters in a character parameter, and has the following characteristics:

Allowed external types
: int (the default), short, unsigned short, unsigned int, long, unsigned long

Allowed PL/SQL types
: VARCHAR2, CHAR, RAW, LONG RAW

Allowed PL/SQL modes
: IN, IN OUT, OUT, RETURN

Call mode
: By value for IN parameters (unless BY REFERENCE specified), and by reference for IN OUT, OUT, and RETURN variables

The LENGTH property is mandatory for RAW and LONG RAW, and is available as a convenience to your C program for the other datatypes in the character family. When passing a RAW from PL/SQL to C, Oracle will set the LENGTH property; however, your C program must set LENGTH if you need to pass the RAW data back.

For an IN parameter, the indicator parameter in your C function might be:

```
int pLenFoo
```

Or for an OUT or IN OUT parameter, it might be:

```
int *pLenFoo
```

The MAXLEN property

The MAXLEN property is an integer indicating the maximum number of characters in a string parameter, and has the following characteristics:

Allowed external types
: int (the default), short, unsigned short, unsigned int, long, unsigned long

Allowed PL/SQL types
: VARCHAR2, CHAR, RAW, LONG RAW

Allowed PL/SQL modes
: IN OUT, OUT, RETURN

Call mode
: By reference

MAXLEN applies to IN OUT, OUT and RETURN parameters and to no other mode. If you attempt to use it for an IN, you'll get the compile-time error *PLS-00250: incorrect usage of MAXLEN in parameters clause*.

Unlike the LENGTH parameter, the MAXLEN data is always passed by reference.

Here's an example of the C formal parameter:

```
int *pMaxLenFoo
```

The CHARSETID and CHARSETFORM properties

The CHARSETID and CHARSETFORM properties are flags denoting information about the character set, and have the following characteristics:

Allowed external types
: unsigned int (the default), unsigned short, unsigned long

Allowed PL/SQL types
: VARCHAR2, CHAR, CLOB

Allowed PL/SQL modes
: IN, IN OUT, OUT, RETURN

Call mode
: By reference

If you are passing data to the external procedure that is expressed in a nondefault character set, these properties will let you communicate the character set's ID and form to the called C program. The values are read-only and should not be modified by the called program. Here is an example of a PARAMETERS clause that includes character set information:

```
PARAMETERS (CONTEXT, cmd STRING, cmd INDICATOR, cmd CHARSETID,
            cmd CHARSETFORM);
```

Oracle sets these additional values automatically, based on the character set in which you have expressed the cmd argument. For more information about Oracle's globalization support in the C program, refer to Oracle's OCI documentation.

Raising an Exception from the Called C Program

The shell( ) program shown earlier in the chapter is very, um, "C-like:" it is a function whose return value contains the status code, and the caller must check the return value to see if it succeeded. Wouldn't it make more sense—in PL/SQL, anyway—for the program to be a procedure that simply raises an exception when there's a problem? Let's take a brief look at how to perform the OCI equivalent of RAISE_APPLICATION_ERROR.

In addition to the easy change from a function to a procedure, there are several other things I need to do:

- Pass in the context area.
- Decide on an error message and an error number in the 20001–20999 range.
- Add a call to the OCI service routine that raises an exception.

The changes to the call spec are trivial:

```
/* File on web: extprocsh.sql */
PROCEDURE shell(cmd IN VARCHAR2)
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocsh"
   WITH CONTEXT
   PARAMETERS (CONTEXT, cmd STRING, cmd INDICATOR);
/
```

(I also removed the return parameter and its indicator.) The following code shows how to receive and use the context pointer in the call needed to raise the exception:

```
/* File on web: extprocsh.c */
#include <ociextp.h>
#include <errno.h>

void extprocsh(OCIExtProcContext *ctx, char *cmd, short cmdInd)
{
   int excNum = 20001;  # a convenient number :->
   char excMsg[512];
   size_t excMsgLen;

    if (cmdInd == OCI_IND_NULL)
       return;

    if (system(cmd) != 0)
    {
       sprintf(excMsg, "Error %i during system call: %.*s", errno, 475,
             strerror(errno));
       excMsgLen = (size_t)strlen(excMsg);

       if (OCIExtProcRaiseExcpWithMsg(ctx, excNum, (text *)excMsg, excMsgLen)
              != OCIEXTPROC_SUCCESS)
          return;
    }

 }
```

Note the following lines:

Line(s)	Description
4	The first of the formal parameters is the context pointer.
6	You can use whatever number in Oracle's user-defined error number range you want; in general, I advise against hardcoding these values, but, er, this is a "do as I say, not as I do" example.
7	The maximum size for the text in a user-defined exception is 512 bytes.
8	A variable to hold the length of the error message text, which will be needed in the OCI call that raises the exception.
10–11	Here, I am translating the NULL argument semantics of the earlier function into a procedure: when called with NULL, nothing happens.
13	A zero return code from system() means that everything executed perfectly; a nonzero code corresponds to either an error or a warning.
15,-17	These lines prepare the variables containing the error message and its length.
19–20	This OCI function, which actually raises the user-defined exception, is where the context pointer actually gets used.

Now, how do we compile this baby? First, Unix/Linux:

```
gcc -m64 -extprocsh.c -I$ORACLE_HOME/rdbms/public -fPIC -shared -o extprocsh.so
```

That was easy enough. But on Microsoft Windows, I found that I needed an explicit *.def* file to define the desired entry point (more precisely, to exclude potential entry points found in Oracle's *oci.lib*):

```
/* File on web: build_extprocsh.bat */
echo LIBRARY extprocsh.dll > extprocsh.def
echo EXPORTS >> extprocsh.def
echo extprocsh >> extprocsh.def
```

Although we've had to break it to fit in the book's margins, the following line must be entered as one long string:

```
c:\MinGW\bin\gcc -c extprocsh.def extprocsh.c -IC:\oracle\product\10.2.0\db_1\oci\
include C:\oracle\product\10.2.0\db_1\oci\lib\msvc\oci.lib-shared -o extprocsh.dll
```

Here's what a test of the function *should* yield:

```
SQL> CALL shell('garbage');
CALL shell('garbage')
     *
ERROR at line 1:
ORA-20001: Error 2 during system call: No such file or directory
```

That is, you should get a user-defined –20001 exception with the corresponding text "no such file or directory." Unfortunately, I discovered that *system()* does not always return meaningful error codes, and on some platforms the message is *ORA-20001: Error 0 during system call: Error 0*. (Fixing this probably requires using a call other than *system()*. Another reader exercise.)

A number of other OCI routines are unique to writing external procedures. Here is the complete list:

OCIExtProcAllocCallMemory
: Allocates memory that Oracle will automatically free when control returns to PL/SQL.

OCIExtProcRaiseExcp
: Raises a predefined exception by its Oracle error number.

OCIExtProcRaiseExcpWithMsg
: Raises a user-defined exception, including a custom error message (illustrated in the previous example).

OCIExtProcGetEnv
: Allows an external procedure to perform OCI callbacks to the database to execute SQL or PL/SQL.

These all require the context pointer. Refer to Oracle's *Application Developer's Guide—Fundamentals* for detailed documentation and examples that use these routines.

Nondefault Agents

Starting with Oracle9*i* Database, it is possible to run external procedure agents via database links that connect to other local database servers. This functionality enables you to spread the load of running expensive external programs onto other database instances.

Even without other servers, running an external procedure through a nondefault agent launches a separate process. This can be handy if you have a recalcitrant external program. Launching it via a nondefault agent means that even if its *extproc* process crashes, it won't have any effect on other external procedures running in the session.

As a simple example of a nondefault agent, here is a configuration that allows an agent to run on the same database but in a separate *extproc* task. The *tnsnames.ora* file needs an additional entry such as:

```
agent0 =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = IPC)(KEY=extprocKey))
    (CONNECT_DATA = (SID = PLSExtProc))
  )
```

Here, *extprocKey* can just be the same key as in your EXTPROC_CONNECTION_DATA entry.

Because agents are created with a database link, we'll need to create one of those:

```
SQL> CREATE DATABASE LINK agent0link
  2  CONNECT TO username IDENTIFIED BY password
  3  USING 'agent0';
```

Now, finally, the agent can appear in a CREATE LIBRARY statement such as:

```
CREATE OR REPLACE LIBRARY extprocshell_lib_with_agent
   AS 'c:\oracle\admin\local\lib\extprocsh.dll'
   AGENT 'agent0';
```

Any call spec that was written to use this library will authenticate and connect through this agent0 link, launching an *extproc* task separate from the default extproc task. In this way, you could separate tasks from each other (for example, you could send the thread-safe tasks to a multithreading agent and the others to dedicated agents).

Oracle also supports a dynamic arrangement that allows you to pass in the name of the agent as a parameter to the external procedure. To take advantage of this feature, use the AGENT IN clause in the call spec. For example (changes in boldface):

```
CREATE OR REPLACE PROCEDURE shell2 (name_of_agent IN VARCHAR2, cmd VARCHAR2)
AS
   LANGUAGE C
   LIBRARY extprocshell_lib
   NAME "extprocsh2"
   AGENT IN (name_of_agent)
   WITH CONTEXT
   PARAMETERS (CONTEXT, name_of_agent STRING, cmd STRING, cmd INDICATOR);
```

Notice that I had to include the name of the agent in the list of parameters. Oracle enforces a rule that every formal parameter must have a corresponding entry in the PARAMETERS clause. So I have to modify my external C library. In my case, I merely added a second entry point, extprocsh2( ), to the library with the following trivial function:

```
void extprocsh2(OCIExtProcContext *ctx, char *agent, char *cmd, short cmdInd)
{
   extprocsh(ctx, cmd, cmdInd);
}
```

My code just ignores the agent string. Now, though, I can invoke my shell2 procedure as in the following:

```
CALL shell2('agent0', 'whatever');
```

If you want your stored program to somehow invoke an external procedure on a remote machine, you have a couple of options. You could implement an external procedure on the local machine which is just a "pass-through" program, making a C-based remote procedure call on behalf of PL/SQL. Alternatively, you could implement a stored PL/SQL program on the remote machine as an external procedure, and call it from the local machine via a database link. What isn't possible is setting up an external procedure listener to accept networked connections from a different machine.

A Debugging Odyssey

It turns out that some debuggers, including the GNU debugger (GDB), can attach to a running process and debug external procedures. Here is an outline of how I got this to work on Solaris and on Windows XP.

As a preliminary step on both platforms, I compiled the shared library file with the compiler option (*-g* in the case of GCC) needed to include symbolic information for the debugger. That was the easy part. During testing, I discovered the slap-hand-on-forehead fact that I could not debug my 64-bit Solaris external procedure with a 32-bit debugger, so I also had to build and install a 64-bit *gdb* executable. This step began in the root directory of the GDB source tree with the command:

```
OS> CC="gcc -m64" ./configure
```

At this point, the GDB build presented many surprises; it probably would have helped to have a competent system administrator looking over my shoulder!

Another preparatory step, although optional, involves running the script *dbgextp.sql*, which should be in the *plsql/demo* directory. If you're using Oracle Database 10*g* or later, you won't find this directory in your default $ORACLE_HOME distribution because Oracle moved the entire directory to the Companion CD. However, you may be able to extract the *plsql/demo* directory with a command like this:

```
OS> jar xvf /cdrom/stage/Components/oracle.rdbms.companion/10. x.x.x.x /1/
DataFiles/filegroup1.jar
```

If you do manage to locate the *dbgextp.sql* file, you'll find that it contains some useful inline comments, which you should definitely read. Then run the script as yourself (not as SYS) to build a package named DEBUG_EXTPROC. This package contains a procedure whose sole purpose in life is to launch the external procedure agent, thus allowing you to discover the corresponding process ID (PID). In a fresh SQL*Plus session, you can run it as follows:

```
SQL> EXEC DEBUG_EXTPROC.startup_extproc_agent
```

This causes an *extproc* process to launch; its PID can be found using *ps -ef* or *pgrep extproc*.

Why do I say the DEBUG_EXTPROC package is optional? Because you can also launch the agent by running any old external procedure; or, if you happen to be using multi-threaded agents, the process will already be pre-spawned, and you can probably figure out the PID without breaking a sweat.

At any rate, armed with the PID, you can start the debugger and attach to the running process:

```
OS> gdb $ORACLE_HOME/bin/extproc pid
```

When I first tried this, I got a "permission denied" error, which was cured by logging in as the *oracle* account.

I then set a breakpoint on the "pextproc" symbol, per the instructions in the file *dbgextp.sql*. Next, in my SQL*Plus session, I invoked my external procedure using:

```
SQL> CALL shell(NULL);
```

I issued a GDB "continue," and *extproc* promptly hit the pextproc breakpoint. Next, I executed a GDB "share" command so the debugger would read the symbols in my just-loaded external shared library; and, finally, I was able to set a breakpoint on the *extprocsh()* external procedure, issue a "continue," and boom—I'm in my code! It worked pretty well after that, allowing me to step through each line of my code, examine variables, etc.

I found that debugging external procedures with Cygwin's GDB on Microsoft platforms required the following adjustments:

- I had to modify the listener service in the Windows control panel to execute under the authority of my own user account rather than under that of "Local System."
- Instead of *ps -ef*, I used Microsoft's *tasklist.exe* program (or the Windows task manager) to obtain the *extproc* PID.
- To view the external procedure's source code during the debugging session, I found that I needed to launch GDB from the directory containing its source file (there is probably another way to do this).

On Solaris, my tests were performed using a 64-bit build of GDB 6.3 on Solaris 2.8. On my Windows XP machine, I used Cygwin's GDB 6.3 binary with no problems, but was not able to get the MinGW GDB 5.2.1 binary to work.

Maintaining External Procedures

Here are some assorted bits of information that will assist you in creating, debugging, and managing external procedures.

Dropping Libraries

The syntax for dropping a library is simply:

```
DROP LIBRARY library_name;
```

The Oracle user who executes this command must have the DROP LIBRARY or DROP ANY LIBRARY privilege.

Oracle does not check dependency information before dropping the library. This fact is useful if you need to change the name or location of the shared object file to which the library points. You can just drop it and rebuild it, and any dependent routines will continue to function. (More useful, perhaps, would be a requirement that you use a DROP LIBRARY FORCE command, but such an option does not exist.)

Before you drop the library permanently, you may wish to look in the DBA_DEPENDENCIES view to see if any PL/SQL module relies on the library.

Data Dictionary

There are a few entries in the data dictionary that help manage external procedures. Table 28-3 shows the USER_ version of the dictionary tables, but note that there are corresponding entries for DBA_ and ALL_.

Table 28-3. Data dictionary views for external procedures

To answer the question...	Use this view	Example
What libraries have I created?	USER_LIBRARIES	`SELECT * FROM user_libraries;`
What stored PL/SQL programs use the xyz library in a call spec?	USER_DEPENDENCIES	`SELECT * FROM user_dependencies WHERE referenced_name = 'XYZ';`
What external procedure agents (both dedicated and multithreaded) are currently running?	V$HS_AGENT	`SELECT * FROM V$HS_AGENT WHERE UPPER(program) LIKE 'EXTPROCS%'`
What Oracle sessions are using which agents?	V$HS_SESSION	`SELECT s.username, h.agent_id FROM V$SESSION s, V$HS_SESSION h WHERE s.sid = h.sid;`

Rules and Warnings

As with almost all things PL/SQL, external procedures come with an obligatory list of cautions:

- While the mode of each formal parameter (IN, IN OUT, OUT) may have certain restrictions in PL/SQL, C does not honor these modes. Differences between the PL/SQL parameter mode and the usage in the C module cannot be detected at compile time, and could also go undetected at runtime. The rules are what you would expect: don't assign values to IN parameters, don't read OUT parameters; always assign values to IN OUT and OUT parameters, and always return a value of the appropriate datatype.
- Modifiable INDICATORs and LENGTHs are always passed by reference for IN OUT, OUT, and RETURN. Unmodifiable INDICATORs and LENGTHs are always passed by value unless you specify BY REFERENCE. However, even if you pass INDICATORs or LENGTHs for PL/SQL variables by reference, they are still read-only parameters.
- Although you can pass up to 128 parameters between PL/SQL and C, if any of them are float or double, your actual maximum will be lower. How much lower depends on the operating system.
- If you use the multithreaded agent feature introduced in Oracle Database 10g, there are special additional restrictions on your programs. All the calls you invoke from the C program must be thread-safe. In addition, you want to avoid using global C variables. Even in the nonthreaded version, globals may not behave as expected due to "DLL caching" by the operating system.

- Your external procedure may not perform DDL commands, begin or end a session, or control a transaction using COMMIT or ROLLBACK. (See Oracle's *PL/SQL User's Guide and Reference* for a list of unsupported OCI routines.)