7
Calling Stored Procedures

After you load and publish a Java stored procedure, you can call it. This chapter describes the procedure for calling Java stored procedures in various contexts. It also describes how Oracle JVM handles SQL exceptions.

This chapter contains the following sections:

- Calling Java from the Top Level
- Calling Java from Database Triggers
- Calling Java from SQL DML
- Calling Java from PL/SQL
- Calling PL/SQL from Java
- How Oracle JVM Handles Exceptions

7.1 Calling Java from the Top Level

The SQL `CALL` statement lets you call Java methods, which are published at the top level, in PL/SQL packages, or in SQL object types. In SQL*Plus, you can run the `CALL` statement interactively using the following syntax:

```
CALL [schema_name.][{package_name | object_type_name}][@dblink_name]
  { procedure_name ([param[, param]...])
   | function_name ([param[, param]...]) INTO :host_variable};
```

where `param` is represented by the following syntax:

```
{literal | :host_variable}
```

Host variables are variables that are declared in a host environment. They must be prefixed with a colon. The following examples show that a host variable cannot appear twice in the same `CALL` statement and that a subprogram without parameters must be called with an empty parameter list:

```
CALL swap(:x, :x); -- illegal, duplicate host variables
CALL balance() INTO :current_balance; -- () required
```

This section covers the following topics:

- Redirecting the Output
- Examples of Calling Java Stored Procedures From the Top Level

7.1.1 Redirecting the Output

On the server, the default output device is a trace file and not the user screen. As a result, `System.out` and `System.err` print output to the current trace files. To redirect output to the SQL*Plus text buffer, you must call the `set_output()` procedure in the `DBMS_JAVA` package, as follows:

```
SQL> SET SERVEROUTPUT ON
SQL> CALL dbms_java.set_output(2000);
```

The minimum buffer size is 2,000 bytes, which is also the default size, and the maximum buffer size is 1,000,000 bytes. In the following example, the buffer size is increased to 5,000 bytes:

```
SQL> SET SERVEROUTPUT ON SIZE 5000
SQL> CALL dbms_java.set_output(5000);
```

The output is displayed when the stored procedure exits.

7.1.2 Examples of Calling Java Stored Procedures From the Top Level

This section provides the following examples

- Example 7-1
- Example 7-2

Example 7-1 A Simple JDBC Stored Procedure

In the following example, the `main()` method accepts the name of a database table, such as `employees`, and an optional `WHERE` clause specifying a condition, such as `salary > 1500`. If you omit the condition, then the method deletes all rows from the table, else it deletes only those rows that meet the condition.

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

public class Deleter
{
  public static void main (String[] args) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    String sql = "DELETE FROM " + args[0];
    if (args.length > 1)
      sql += " WHERE " + args[1];
    try
    {
      Statement stmt = conn.createStatement();
      stmt.executeUpdate(sql);
      stmt.close();
    }
    catch (SQLException e)
    {
      System.err.println(e.getMessage());
    }
  }
}
```

The `main()` method can take either one or two arguments. Usually, the `DEFAULT` clause is used to vary the number of arguments passed to a PL/SQL subprogram. However, this clause is not allowed in a call specification. As a result, you must overload two packaged procedures, as follows:

```
CREATE OR REPLACE PACKAGE pkg AS
PROCEDURE delete_rows (table_name VARCHAR2);
PROCEDURE delete_rows (table_name VARCHAR2, condition VARCHAR2);
END;

CREATE OR REPLACE PACKAGE BODY pkg AS
PROCEDURE delete_rows (table_name VARCHAR2)
```

```
AS LANGUAGE JAVA
NAME 'Deleter.main(java.lang.String[])';

PROCEDURE delete_rows (table_name VARCHAR2, condition VARCHAR2)
AS LANGUAGE JAVA
NAME 'Deleter.main(java.lang.String[])';
END;
```

Now, you can call the `delete_rows` procedure, as follows:

```
SQL> CALL pkg.delete_rows('employees', 'salary > 1500');

Call completed.

SQL> SELECT first_name, salary FROM employees;

FIRST_NAME  SALARY
--------- --------
SMITH          800
WARD          1250
MARTIN        1250
TURNER        1500
ADAMS         1100
JAMES          950
MILLER        1300

7 rows selected.
```

> **Note:**
>
> You cannot overload top-level procedures.

Example 7-2 Fibonacci Sequence

Assume that the executable for the following Java class is stored in Oracle Database:

```
public class Fibonacci
{
  public static int fib (int n)
  {
    if (n == 1 || n == 2)
      return 1;
    else
      return fib(n - 1) + fib(n - 2);
  }
}
```

The `Fibonacci` class has a method, `fib()`, which returns the *n*th Fibonacci number. The Fibonacci sequence, 1, 1, 2, 3, 5, 8, 13, 21, . . ., is recursive. Each term in the sequence, after the second term, is the sum of the two terms that immediately precede it. Because `fib()` returns a value, you must publish it as a function, as follows:

```
CREATE OR REPLACE FUNCTION fib (n NUMBER) RETURN NUMBER
AS LANGUAGE JAVA
NAME 'Fibonacci.fib(int) return int';
```

Next, you declare two SQL*Plus host variables and initialize the first one:

```
SQL> VARIABLE n NUMBER
SQL> VARIABLE f NUMBER
SQL> EXECUTE :n := 7;

PL/SQL procedure successfully completed.
```

Now, you can call the `fib()` function. In a `CALL` statement, host variables must be prefixed with a colon. The function can be called, as follows:

```
SQL> CALL fib(:n) INTO :f;

Call completed.

SQL> PRINT f

F
----------
13
```

7.2 Calling Java from Database Triggers

A database trigger is a stored program that is associated with a specific table or view. Oracle Database runs the trigger automatically whenever a data manipulation language (DML) operation affects the table or view.

When a triggering event occurs, the trigger runs and either a PL/SQL block or a `CALL` statement performs the action. A statement trigger runs once, before or after the triggering event. A row trigger runs once for each row affected by the triggering event.

In a database trigger, you can reference the new and old values of changing rows by using the correlation names `new` and `old`. In the trigger-action block or `CALL` statement, column names must be prefixed with `:new` or `:old`.

The following are examples of calling Java stored procedures from a database trigger:

- Example 7-3
- Example 7-4

Example 7-3 Calling Java Stored Procedure from Database Trigger - I

Assume you want to create a database trigger that uses the following Java class to log out-of-range salary increases:

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

public class DBTrigger
{
  public static void logSal (int empID, float oldSal, float newSal)
                                                          throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    String sql = "INSERT INTO sal_audit VALUES (?, ?, ?)";
    try
    {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, empID);
      pstmt.setFloat(2, oldSal);
      pstmt.setFloat(3, newSal);
      pstmt.executeUpdate();
```

```
      pstmt.close();
    }
    catch (SQLException e)
    {
      System.err.println(e.getMessage());
    }
  }
}
```

The `DBTrigger` class has one method, `logSal()`, which inserts a row into the `sal_audit` table. Because `logSal()` is a `void` method, you must publish it as a procedure:

```
CREATE OR REPLACE PROCEDURE log_sal (
 emp_id NUMBER,
 old_sal NUMBER,
 new_sal NUMBER
)
AS LANGUAGE JAVA
NAME 'DBTrigger.logSal(int, float, float)';
```

Next, create the `sal_audit` table, as follows:

```
CREATE TABLE sal_audit (
 empno NUMBER,
 oldsal NUMBER,
 newsal NUMBER
);
```

Finally, create the database trigger, which fires when a salary increase exceeds 20 percent:

```
CREATE OR REPLACE TRIGGER sal_trig
AFTER UPDATE OF salary ON employees
FOR EACH ROW
WHEN (new.salary > 1.2 * old.salary)
CALL log_sal(:new.employee_id, :old.salary, :new.salary);
```

When you run the following `UPDATE` statement, it updates all rows in the `employees` table:

```
SQL> UPDATE employee SET salary = salary + 300;
```

For each row that meets the condition set in the `WHEN` clause of the trigger, the trigger runs and the Java method inserts a row into the `sal_audit` table.

```
SQL> SELECT * FROM sal_audit;

     EMPNO     OLDSAL     NEWSAL
---------- ---------- ----------
      7369        800       1100
      7521       1250       1550
      7654       1250       1550
      7876       1100       1400
      7900        950       1250
      7934       1300       1600

6 rows selected.
```

Example 7-4 Calling Java Stored Procedure from Database Trigger - II

Assume you want to create a trigger that inserts rows into a database view, which is defined as follows:

```
CREATE VIEW emps AS
SELECT empno, ename, 'Sales' AS dname FROM sales
```

```
UNION ALL
SELECT empno, ename, 'Marketing' AS dname FROM mktg;
```

The `sales` and `mktg` database tables are defined as:

```
CREATE TABLE sales (empno NUMBER(4), ename VARCHAR2(10));
CREATE TABLE mktg (empno NUMBER(4), ename VARCHAR2(10));
```

You must write an `INSTEAD OF` trigger because rows cannot be inserted into a view that uses set operators, such as `UNION ALL`. Instead, the trigger will insert rows into the base tables.

First, add the following Java method to the `DBTrigger` class, which is defined in Example 7-3:

```
public static void addEmp (int empNo, String empName, String deptName)
                                                              throws SQLException
{
  Connection conn = DriverManager.getConnection("jdbc:default:connection:");
  String tabName = (deptName.equals("Sales") ? "sales" : "mktg");
  String sql = "INSERT INTO " + tabName + " VALUES (?, ?)";
  try
  {
    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setInt(1, empNo);
    pstmt.setString(2, empName);
    pstmt.executeUpdate();
    pstmt.close();
  }
  catch (SQLException e)
  {
    System.err.println(e.getMessage());
  }
}
```

The `addEmp()` method inserts a row into the `sales` or `mktg` table depending on the value of the `deptName` parameter. Write the call specification for this method, as follows:

```
CREATE OR REPLACE PROCEDURE add_emp (
 emp_no NUMBER,
 emp_name VARCHAR2,
 dept_name VARCHAR2
)
AS LANGUAGE JAVA
NAME 'DBTrigger.addEmp(int, java.lang.String, java.lang.String)';
```

Next, create the `INSTEAD OF` trigger, as follows:

```
CREATE OR REPLACE TRIGGER emps_trig
INSTEAD OF INSERT ON emps
FOR EACH ROW
CALL add_emp(:new.empno, :new.ename, :new.dname);
```

When you run each of the following `INSERT` statements, the trigger runs and the Java method inserts a row into the appropriate base table:

```
SQL> INSERT INTO emps VALUES (8001, 'Chand', 'Sales');
SQL> INSERT INTO emps VALUES (8002, 'Van Horn', 'Sales');
SQL> INSERT INTO emps VALUES (8003, 'Waters', 'Sales');
SQL> INSERT INTO emps VALUES (8004, 'Bellock', 'Marketing');
SQL> INSERT INTO emps VALUES (8005, 'Perez', 'Marketing');
SQL> INSERT INTO emps VALUES (8006, 'Foucault', 'Marketing');

SQL> SELECT * FROM sales;
```

```
     EMPNO ENAME
---------- ----------
      8001 Chand
      8002 Van Horn
      8003 Waters

SQL> SELECT * FROM mktg;

     EMPNO ENAME
---------- ----------
      8004 Bellock
      8005 Perez
      8006 Foucault

SQL> SELECT * FROM emps;

     EMPNO ENAME      DNAME
---------- ---------- ---------
      8001 Chand      Sales
      8002 Van Horn   Sales
      8003 Waters     Sales
      8004 Bellock    Marketing
      8005 Perez      Marketing
      8006 Foucault   Marketing
```

7.3 Calling Java from SQL DML

If you publish Java methods as functions, then you can call them from SQL `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CALL`, `EXPLAIN PLAN`, `LOCK TABLE`, and `MERGE` statements. For example, assume that the executable for the following Java class is stored in Oracle Database:

```
public class Formatter
{
  public static String formatEmp (String empName, String jobTitle)
  {
    empName = empName.substring(0,1).toUpperCase() +
                                     empName.substring(1).toLowerCase();
    jobTitle = jobTitle.toLowerCase();
    if (jobTitle.equals("analyst"))
      return (new String(empName + " is an exempt analyst"));
    else
      return (new String(empName + " is a non-exempt " + jobTitle));
  }
}
```

The `Formatter` class has the `formatEmp()` method, which returns a formatted string containing a staffer's name and job status. Write the call specification for this method, as follows:

```
CREATE OR REPLACE FUNCTION format_emp (ename VARCHAR2, job VARCHAR2)
RETURN VARCHAR2
AS LANGUAGE JAVA
NAME 'Formatter.formatEmp (java.lang.String, java.lang.String)
return java.lang.String';
```

Now, call the `format_emp` function to format a list of employees:

```
SQL> SELECT format_emp(first_name, job_id) AS "Employees" FROM employees
  2   WHERE job_id NOT IN ('AC_MGR', 'AD_PRES') ORDER BY first_name;

Employees
```

```
-----------------------------------------
Adams is a non-exempt clerk
Allen is a non-exempt salesman
Ford is an exempt analyst
James is a non-exempt clerk
Martin is a non-exempt salesman
Miller is a non-exempt clerk
Scott is an exempt analyst
Smith is a non-exempt clerk
Turner is a non-exempt salesman
Ward is a non-exempt salesman
```

Restrictions

A Java method must adhere to the following rules, which are meant to control side effects:

- When you call a method from a `SELECT` statement or parallel `INSERT`, `UPDATE`, or `DELETE` statements, the method cannot modify any database tables.
- When you call a method from an `INSERT`, `UPDATE`, or `DELETE` statement, the method cannot query or modify any database tables modified by that statement.
- When you call a method from a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the method cannot run SQL transaction control statements, such as `COMMIT`, session control statements, such as `SET ROLE`, or system control statements, such as `ALTER SYSTEM`. In addition, the method cannot run data definition language (DDL) statements, such as `CREATE`, because they are followed by an automatic commit.

If any SQL statement inside the method violates any of the preceding rules, then you get an error at run time.

7.4 Calling Java from PL/SQL

You can call Java stored procedures from any PL/SQL block, subprogram, or package. For example, assume that the executable for the following Java class is stored in Oracle Database:

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

public class Adjuster
{
  public static void raiseSalary (int empNo, float percent) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    String sql = "UPDATE employees SET salary = salary * ? WHERE employee_id = ?";
    try
    {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setFloat(1, (1 + percent / 100));
      pstmt.setInt(2, empNo);
      pstmt.executeUpdate();
      pstmt.close();
    }
    catch (SQLException e)
    {
      System.err.println(e.getMessage());
    }
  }
}
```

The `Adjuster` class has one method, which raises the salary of an employee by a given percentage. Because `raiseSalary()` is a `void` method, you must publish it as a procedure, as follows:

```
CREATE OR REPLACE PROCEDURE raise_salary (empno NUMBER, pct NUMBER)
AS LANGUAGE JAVA
NAME 'Adjuster.raiseSalary(int, float)';
```

In the following example, you call the `raise_salary` procedure from an anonymous PL/SQL block:

```
DECLARE
emp_id NUMBER;
percent NUMBER;
BEGIN
-- get values for emp_id and percent
raise_salary(emp_id, percent);
...
END;
```

In the following example, you call the `row_count` function, which defined in Example 6-3, from a standalone PL/SQL stored procedure:

```
CREATE PROCEDURE calc_bonus (emp_id NUMBER, bonus OUT NUMBER) AS
emp_count NUMBER;
...
BEGIN
emp_count := row_count('employees');
...
END;
```

In the following example, you call the `raise_sal` method of the `Employee` object type, which is defined in "Implementing Object Type Methods", from an anonymous PL/SQL block:

```
DECLARE
emp_id NUMBER(4);
v emp_type;
BEGIN
-- assign a value to emp_id
SELECT VALUE(e) INTO v FROM emps e WHERE empno = emp_id;
v.raise_sal(500);
UPDATE emps e SET e = v WHERE empno = emp_id;
...
END;
```

7.5 Calling PL/SQL from Java

Java Database Connectivity (JDBC) enable you to call PL/SQL stored functions and procedures. For example, you want to call the following stored function, which returns the balance of a specified bank account:

```
FUNCTION balance (acct_id NUMBER) RETURN NUMBER IS
acct_bal NUMBER;
BEGIN
SELECT bal INTO acct_bal FROM accts
WHERE acct_no = acct_id;
RETURN acct_bal;
END;
```

In a JDBC program, a call to the `balance` function can be written as follows:

```
...
CallableStatement cstmt = conn.prepareCall("{? = CALL balance(?)}");
cstmt.registerOutParameter(1, Types.FLOAT);
cstmt.setInt(2, acctNo);
cstmt.executeUpdate();
float acctBal = cstmt.getFloat(1);
...
```

7.6 How Oracle JVM Handles Exceptions

Java exceptions are objects and have a naming and inheritance hierarchy. As a result, you can substitute a subexception, that is, a subclass of an exception class, for its superexception, that is, the superclass of an exception class.

All Java exception objects support the `toString()` method, which returns the fully qualified name of the exception class concatenated to an optional string. Typically, the string contains data-dependent information about the exceptional condition. Usually, the code that constructs the exception associates the string with it.

When a Java stored procedure runs a SQL statement, any exception thrown is materialized to the procedure as a subclass of `java.sql.SQLException`. This class has the `getErrorCode()` and `getMessage()` methods, which return the Oracle error code and message, respectively.

If a stored procedure called from SQL or PL/SQL throws an exception and is not caught by Java, then the following error message appears:

```
ORA-29532 Java call terminated by uncaught Java exception
```

This is how all uncaught exceptions, including non-SQL exceptions, are reported.