

# CallableStatement handling using named parameters

## A new feature for Informix Dynamic Server

Skill Level: Intermediate

[Akhilesh K. Tiwary \(aktiwary@in.ibm.com\)](mailto:aktiwary@in.ibm.com)  
Staff Software Engineer, Informix CSDK Lead  
IBM

[Thamizhchelvan A. Anbalagan \(tanbalag@in.ibm.com\)](mailto:tanbalag@in.ibm.com)  
System Software Engineer, Development Engineer  
IBM

21 Feb 2008

Explore a powerful new feature of IBM® Informix® Dynamic Server (IDS) called "named parameters in a CallableStatement," which enables you to identify a parameter using its name instead of its ordinal position. This feature thus extends the capability of using CallableStatements so that they can be used either by ordinal position or parameter name. Learn the difference between the two techniques, and see the advantages of named parameters over the ordinal technique.

## Introduction: Statement handling in JDBC

In a JDBC application, a JDBC statement object is used to send an SQL statement to the database server. A statement object is associated with a connection, and it is the statement object that handles communication between application and database server.

There are three types of statement objects available in JDBC:

1. [General statement](#)

2. [Prepared statement](#)
3. [Callable statement](#)

Statement objects are associated with a connection, so you should establish a database connection to create a statement object.

## Creating a connection

The code sample in Listing 1, below, illustrates how to create a connection:

### Listing 1. Sample code to load Informix driver and create a connection

```
Connection con = null;
try {
    Class.forName("com.informix.jdbc.IfxDriver");
    String url = "jdbc:informix-sqli://hostname:port_number/dbname:
informixserver=servername; userid=userid;password=pwd;";
    con = DriverManager.getConnection(url);
}
```

Let's now examine the three types of statement object, one by one.

## General statements

A connection's `createStatement` method is used to create this statement. It is specifically used for the SQL statements where you don't need to pass any value as a parameter.

### Listing 2. Sample code illustrating the create statement

```
Statement stmt = con.createStatement();
    cmd = "create database testDB;";
    rc = stmt.executeUpdate(cmd);
stmt.close();
```

## Prepared statements

Prepared statements are a subclass of the statement class. The main difference is that, unlike the statement class, prepared statement is compiled and optimized once and can be used multiple times by setting different parameter values. So when you want to execute a statement many times, prepared statement is a better option. As it is in compiled form, it reduces the execution time. So the benefit is that prepared statement not only contains an SQL statement, but a pre-compiled SQL statement. Another difference is that the SQL statement is given to the prepared statement right

when it is created.

### Listing 3. Sample code to explain the prepared statement

```
PreparedStatement pstmt = con.prepareStatement("UPDATE tabl "+
      "set col1 = ? where key = 1");
pstmt.setShort(1, (short)2);
int rowcount = pstmt.executeUpdate();
```

Here, the same prepared statement can be used for another set of value for col1. Once a parameter has been set, it retains that value until it is reset or `clearParameters` is called. This feature enables prepared statements to be used for batch INSERT/UPDATE.

### Batch update

The batch update feature improves the performance of a single statement that is executed multiples times, with multiple value settings. This allows multiple update operations to be submitted to a data source for processing at once. Batch update can be used by the statement object also. But a statement object submits different SQL statements to batch, while prepared statement submits a set of parameters to batch.

Listing 4 shows how to use batch insert with a prepared statement:

### Listing 4. Sample code illustrating batch update

```
PreparedStatement pst = conn.prepareStatement("insert into tabl values (?)");
for loop....
{
    pst.setInt (1, i);
    pst.addBatch();
}
pst.executeBatch();
```

The `addBatch` method adds the statement to a cache and flushes to the database using the `executeBatch()` method. So it saves compilation / optimization of the statement, as it is compiled only once (with prepared statement), and also saves a round trip to the server, as it sends the batch insert at one shot.

## Callable statement

This is the third way of calling an SQL statement, and it provides a way to call a stored procedure on the server from a Java™ program. Callable statements also need to be prepared first, and then their parameters are set using the set methods. Parameters value can be set by either of two ways:

1. Ordinal position
2. Named parameter

Ordinal position is the traditional way of setting the parameter by its position in the `CallableStatements`. But named parameter provides the flexibility to set the parameters by name, instead of by ordinal position. Parameters for the `CallableStatement` must be specified by either name or by the ordinal format within a single invocation of a routine. If you name a parameter for one argument, for example, you must use parameter names for all of the arguments.

Named parameters are especially useful for calling stored procedures that have many arguments and some of those arguments have default values. If the procedure is unique, you can omit parameters that have default values and enter the parameters in any order. Named parameter makes the application robust so that you don't need to change the application even if the sequence of the parameter in the stored procedure changes.

The JDBC driver provides the

`DatabaseMetaData.supportsNamedParameters()` method to determine if the driver and the RDBMS support named parameters in a `CallableStatement`. The system returns true if named parameters are supported. For example:

#### Listing 5. Usage of `supportsNamedParameters()`

```

Connection myConn = . . . // connection to the RDBMS for Database
DatabaseMetaData dbmd = myConn.getMetaData();
if (dbmd.supportsNamedParameters() == true)
{
    System.out.println("NAMED PARAMETERS FOR CALLABLE"
        + " STATEMENTS IS SUPPORTED");
}

```

#### Retrieving parameter names for stored procedures

Names of parameters for stored procedures can be retrieved using `getProcedureColumns` of `DatabaseMetaData`, defined as shown in Listing 6:

#### Listing 6. Usage of `getProcedureColumn()` method

```

Connection myConn = . . . // connection to the RDBMS for Database
. . .
DatabaseMetaData dbmd = myConn.getMetaData();
ResultSet rs = dbmd.getProcedureColumns(
    "myDB", schemaPattern, procedureNamePattern, columnNamePattern);
rs.next() {
    String parameterName = rs.getString(4);
- - - or - - -
    String parameterName = rs.getString("COLUMN_NAME");
}

```

```

- - -
System.out.println("Column Name: " + parameterName);

```

The names of all columns that match the parameters of the `getProcedureColumns()` method will be displayed.

Listing 7 shows the uses of named parameter for `CallableStatements`.

## Create a stored procedure

### Listing 7. Usage of callable OUT parameter

```

create procedure createProductDef(productname  varchar(64),
                                productdesc  varchar(64),
                                listprice    float,
                                minprice     float,
                                out prod_id   float);
. . .
let prod_id="value for prod_id";
end procedure;

```

The Java code in [Listing 8](#) first creates a `CallableStatement` with five parameters that correspond to the stored procedure. The question mark characters (?) within the parentheses of a JDBC call refer to the parameters. Set or register all the parameters. Name the parameters using the format `cstmt.setString("arg", name);`, where `arg` is the name of the argument in the corresponding stored procedure. You do not need to name parameters in the same order as the arguments in the stored procedure.

### Listing 8. Usage of callable named parameter

```

String sqlCall = "{call CreateProductDef(?,?,?,?,?)}";
CallableStatement cstmt = conn.prepareCall(sqlCall);

cstmt.setString("productname", name);      // Set Product Name.
cstmt.setString("productdesc", desc);     // Set Product Description.
cstmt.setFloat("listprice", listprice);   // Set Product ListPrice.
cstmt.setFloat("minprice", minprice);     // Set Product MinPrice.

// Register out parameter which should return the product is created.

cstmt.registerOutParameter("prod_id", Types.FLOAT);

// Execute the call.
cstmt.execute();

// Get the value of the id from the OUT parameter: prod_id
float id = cstmt.getFloat("prod_id");

```

If the number of parameters in `CallableStatement` is less than the number of arguments in the stored procedure, the remaining arguments must have default values. You do not need to set values for arguments that have default values

because the server automatically uses the default values. For example, if a stored procedure has 10 arguments of which four have non-default values and six have default values, you must have at least four question marks in the CallableStatement. Alternatively, you can use five, six, or up to 10 question marks. In the following unique stored procedure the arguments `listprice` and `minprice` have default values:

### Listing 9. Create a procedure with arguments having default values

```
create procedure createProductDef(productname  varchar(64),
                                productdesc  varchar(64),
                                listprice    float default 100.00,
                                minprice     float default  90.00,
                                out prod_id   float);
.
.
.
let prod_id = value for prod_id;
end procedure;
```

The Java code in Listing 10, below, calls the stored procedure with fewer parameters than arguments in the stored procedure (four parameters for five arguments). Because `listprice` has a default value, it can be omitted from the CallableStatement.

### Listing 10. Usage of default parameters

```
String sqlCall = "{call CreateProductDef(?,?,?,?)}";
// 4 params for 5 args
CallableStatement cstmt = conn.prepareCall(sqlCall);

cstmt.setString("productname", name); // Set Product Name.
cstmt.setString("productdesc", desc); // Set Product Description.

cstmt.setFloat("minprice", minprice); // Set Product MinPrice.

// Register out parameter which should return the product id created.
cstmt.registerOutParameter("prod_id", Types.FLOAT);

// Execute the call.
cstmt.execute();

// Get the value of the id from the OUT parameter: prod_id
float id = cstmt.getFloat("prod_id");
```

If a callable statement contains OUT or INOUT parameter, then you need to register those parameters using `registerOutParameter` of the CallableStatement. Listing 11 uses the out parameter `prod_id` for creating a stored procedure with the OUT parameter. Similarly, the INOUT parameter can be created using the keyword INOUT.

### Listing 11. Usage of INOUT and OUT parameter

```

create procedure createProductDef(productname  varchar(64),
                                productdesc  varchar(64),
                                inout  listprice  float default 100.00,
                                minprice  float default 90.00,
                                out prod_id  float);

```

Listing 12 uses CallableStatements registerOutparameter method to register the out parameter of the CallableStatement.

### Listing 12. Registering OUT parameter with CallableStatement

```

cstmt.registerOutParameter("prod_id", Types.FLOAT);

```

Listing 13 consolidates all statements with the functionality of named parameter feature:

### Listing 13. Demo program for named parameter functionality

```

package Callable;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
public class out1 {
    static Connection conn;
    public static void main(String[] args) {
        getConnect();
        System.out.println("Connection Established");
        createProc();
        runthis();
        System.out.println("\n=====Finished=====");
        System.exit(0);
    }
    private static void getConnect() {
        try
        {
            Class.forName("com.informix.jdbc.IfxDriver");
            String url = "jdbc:informix-sqli://host name or ip :port number/database
                name:informixserver=dbservername;";
            System.out.println("URL: "+url);
            conn = DriverManager.getConnection(url);
        }
        catch( Exception e )
        {
            e.printStackTrace();
            System.exit(1);
        }
    }
    private static void createProc() {
        String str=null;
        Statement stmt = null;
        try
        {
            stmt = conn.createStatement();
        }
    }
}

```

```

        catch (SQLException e2)
        {
            e2.printStackTrace();
        }
        str="drop function c_out_proc";
        try
        {
            stmt.executeUpdate (str);
        }
        catch (SQLException e1)
        {
        }
        str = "create function  c_out_proc ( i int, OUT d varchar(20) ) \n" +
        "returning float; \n" +
        "define f float; \n" +
        "let d= \"Hello OUT\"; \n" +
        "let f=i*2; \n" +
        "return f; \n" +
        "end function; \n";
        try
        {
            stmt.executeUpdate (str);
            System.out.println("Function created \n");
        }
        catch (SQLException e)
        {
            System.out.println("Error on creating function: " + e.toString());
            System.exit(1);
        }
    }
}
private static void runthis()
{
    CallableStatement cstmt = null;
    String command = "{? = call c_out_proc(?, ?)} ";
    try
    {
        cstmt = conn.prepareCall (command);
        cstmt.setInt(1, 2);
        cstmt.registerOutParameter(2, Types.VARCHAR);
        ResultSet rs = cstmt.executeQuery();
        if (rs == null)
        {
            System.out.println("rs is null *** this is BAD.");
            System.exit(0);
        }
        else
        {
            rs.next();
            System.out.println(rs.getFloat(1));
            System.out.println(cstmt.getString(2));
        }
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}
}
}

```

## Conclusion

### Share this...



Digg



This article began by introducing the various statement types that are available with IDS JDBC Driver. It then described the named parameter feature and discussed how that can be used in a CallableStatement.

This article ended by listing a demo program that uses the named parameter feature with the IDS 11 server. Now you have the tools to try this for yourself and see the benefits of the named parameter feature.

# Resources

## Learn

- [IBM Informix JDBC Driver Programmer's Guide](#) (IBM Informix Dynamic Server v 10 Information Center): Understand all the JDBC java methods and their usages.
- [New Features in Version 11.10 of IBM Informix Dynamic Server](#) (IBM Informix Dynamic Server v 11 Information Center): Learn about the new features that are added with IDS 11.10.
- [developerWorks Information Management zone](#): Learn more about Information Management. Find technical documentation, how-to articles, education, downloads, product information, and more.
- Stay current with [developerWorks technical events and webcasts](#).
- [Technology bookstore](#): Browse for books on these and other technical topics.

## Get products and technologies

- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

## Discuss

- [Participate in the discussion forum for this content](#).
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

## About the authors

Akhilesh K. Tiwary

Akhilesh Kumar Tiwary is a lead of the CSDK team and works on JDBC in IBM Software Labs, India.

---

Thamizhchelvan A. Anbalagan

Tamil is the Development Engineer for the IBM Informix JDBC team at IBM Software Labs, India. For the last three years, he has been working on JDBC and OAT (Open Admin tool) for IDS.