Advanced database operations with JDBC

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial is designed to introduce you to several advanced database operations, including stored procedures and advanced datatypes, that can be performed by a Java application using JDBC. The example code in this tutorial was written to work with DB2 Universal Database 7.2, but modifying the code to work with other databases is trivial due to the use of a **`DataSource`** object.

This tutorial assumes that you are already familiar with the Java programming language and, to some extent, JDBC. To fully leverage some of the material contained in this tutorial we recommend that you complete the tutorial *Managing database connections with JDBC*. The links in Resources on page 24 include referrals to additional information on JDBC.

What is this tutorial about?

This tutorial demonstrates how to perform advanced operations with a database using JDBC. It focuses on more advanced database functionality, including database design, prepared statements, stored procedures, and advanced datatypes.

The tutorial begins with a discussion of the design of a fictitious message board system. Topics include creating the tables, populating them with data, and using a **`DataSource`** object to ensure maximum portability between databases.

Next, the **`PreparedStatement`** object is introduced along with examples to show how it simplifies certain types of application development, including the population of tables.

The topic of stored procedures is then broached, from how to create Java stored procedures, to the proper handling of stored procedures from an application using the JDBC **`CallableStatement`**.

After this, the **`Blob`** and **`Clob`** advanced datatypes are introduced along with several examples that demonstrate their use.

Tools

Numerous code snippets are provided to help you make the transition from theory to practical use. To actively work through these examples, the following tools need to be installed and working correctly:

* A text editor: Java source files are simply text, so to create and read them, all you need is a text editor. If you have access to a Java IDE, you can also use it, but sometimes IDEs hide too many of the details.

* A Java development environment, such as the Java 2 SDK, which is available at *http://java.sun.com/j2se/1.4/*. The Java 2 SDK, Standard Edition version 1.4, includes the JDBC standard extensions as well as JNDI, which are both necessary for some of the latter examples in the book.

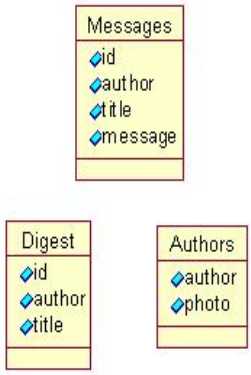
* An SQL-compliant database: The examples in this tutorial use a wide variety of different databases to help demonstrate how database independent JDBC programming can be. Resources on page 24 contains links to more information on both JDBC and databases. In particular, the examples in this tutorial have been tested with DB2 running on a Windows 2000 server, but because they are written using a **`DataSource`** object, they should easily convert to other databases.

* A JDBC driver: Because the JDBC API is predominantly composed of interfaces, you need to obtain an actual JDBC driver implementation to make the examples in this tutorial actually work. The examples in this tutorial use advanced JDBC functionality, and, therefore, they require an advanced JDBC driver. Most database and JDBC driver vendors will supply you with an evaluation version of a particular JDBC driver.

About the author

Robert J. Brunner is an astrophysicist by day and a computer hacker by night. His principal employer is the California Institute of Technology, where he works on knowledge extraction and data-mining from large, highly distributed astronomical databases. He has provided consulting to scientific data centers around the world, provided Java and XML training to a variety of groups, and is currently working on the book *Enterprise Java Database Programming* which will be published by Addison Wesley in 2002. Feel free to contact him via e-mail at *rjbrunner@pacbell.net*.

Section 2. Application design



Database design

This tutorial uses examples based on a fictitious message board application designed to allow multiple authors to post messages, as well as a searchable message digest that summarizes the status of the message board. In a simplified overview of the application, there are three distinct entities:

* A message class
* An author class
* A digest class

Each of these entities will have its own distinct table in the database. This schema is captured in the class diagram to the left, which shows the three classes and their respective attributes.

Initializing the database connection

The code examples in this tutorial use a **`DataSource`** object to obtain a connection to the target database. By default, the target database is DB2 UDB. The first task is to initialize the DB2 **`DataSource`** object and make it available to other applications, which is done using the JNDI filesystem context provider (see Resources on page 24 for additional information on **`DataSource`** objects and JNDI).

Ideally, you initialize a **`DataSource`** object using an administration tool, but it isn't overly difficult to do manually as the following code snippets demonstrate. The full example code is available from Resources on page 24 . To demonstrate how easy it is to change databases, the parts of the code that are database dependent have been marked in boldface.

To begin the connection process, the relevant classes are imported. This includes the actual DataSource implementation used (in this case, the DB2 JDBC **`DataSource`** provided by IBM), as well as the relevant classes for the JNDI naming interface. To use the filesystem context provider, download the appropriate jar file (again, see Resources on page 24 for details).

```
package com.persistentjava;

import COM.ibm.db2.jdbc.DB2DataSource ;

import java.util.Hashtable;
import javax.naming.*;
import javax.naming.directory.*;

import java.sql.* ;
```

```
import javax.sql.DataSource;
```

In the following code segment, one single class is used to create the **DataSource** object and bind it to a name later used to materialize the object from the JNDI filesystem context provider. In this example, the code to initialize the database is made dependent (only in that **db2** is part of the name), but in general the name doesn't need to change, only the actual object bound to the name -- this is the power of name services. Think how DNS works; the actual machine to which you connect might change, but the name stays the same. One last point to note is the last line of code shown below; its purpose is to first unbind (or delete) any object previously bound to the **DataSource** name. While not necessary, this is a good practice because it is a form of resource deallocation.

```
public class InitializeDB2 {
  public static void main(String argv[]) {

    String fsName = "jdbc/pjtutorial/db2";

    try{

      Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.RefFSContextFactory");

      Context ctx = new InitialContext(env);

      ctx.unbind(fsName) ;
```

Creating the DataSource object

To enable the magic of JNDI, we first need to create the **DataSource** object and set all relevant properties. There is a standard list of properties that a **DataSource** object must take, as well as several optional (non-essential) properties. A GUI tool could discover these extra properties through reflection and query the user appropriately. Once the **DataSource** object has been created and initialized, the next step is to bind it to the appropriate name.

```
      DB2DataSource ds = new DB2DataSource();
      ds.setDescription("DB2 DataSource");
      ds.setServerName("persistentjava.com");
      ds.setPortNumber(6789);
      ds.setDatabaseName("jdbc");

      ctx.bind(fsName, ds) ;
      ctx.close() ;

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Section 3. Database creation

Schema creation

While database programming has become considerably easier over the years with the proliferation of GUI design tools, it is still instructive (and fun) to craft a schema by hand. Using JDBC, such creation is easy; we only need to focus on the actual SQL data definition language (DDL) statements required to create the database tables for our application (see Resources on page 24 for links to SQL material). As discussed in the previous section of this tutorial, the message board application employs three classes: the message class, the digest class, and the author class. The following SQL statements create the requisite tables.

```
String dTableSQL =  "CREATE TABLE digest " +
                    "(id INTEGER NOT NULL," +
                    " title VARCHAR(64) NOT NULL," +
                    " author VARCHAR(64) NOT NULL)" ;

String mTableSQL =  "CREATE TABLE messages " +
                    "(id INTEGER NOT NULL," +
                    " title VARCHAR(64) NOT NULL," +
                    " author VARCHAR(64) NOT NULL," +
                    " message CLOB(2048))" ;

String aTableSQL =  "CREATE TABLE authors " +
                    "(author VARCHAR(64) NOT NULL," +
                    " photo BLOB(4096))" ;
```

Given the preceding DDL statements, we can quickly construct a JDBC application to create these tables in the appropriate database. The following code snippet demonstrates this process; the complete example is available in the source code zip file in Resources on page 24 . An ellipsis is used to indicate a section where code has been removed to increase the clarity of the example. Note how the example first materializes the **`DataSource`** object using JNDI and the filesystem provider, creates a new database connection, and executes the SQL DDL statements, which in turn update the database by creating new tables.

```
package com.persistentjava;

... // The SQL Statements go here.

    String fsName = "jdbc/pjtutorial/db2";

    Connection con = null ;

     try {

      Hashtable env = new Hashtable() ;
      env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.RefFSContextFactory") ;


      Context ctx = new InitialContext(env) ;
      DataSource ds = (DataSource)ctx.lookup(fsName) ;
```

```
      con = ds.getConnection("java", "sun") ;
      Statement stmt = con.createStatement() ;


      stmt.executeUpdate(dTableSQL) ;
      stmt.executeUpdate(mTableSQL) ;
      stmt.executeUpdate(aTableSQL) ;


      System.out.println("Tables Created Successfully") ;
...
```

Error handling

The rest of the application focuses on error handling, which is non-trivial when dealing with databases due to the fact SQL exception objects can be chained. First, we handle any potential error problems with the appropriate catch block, and after everything has completed, we close the **`Connection`** in a **`finally`** block. However, because the connection close method can throw an **`SQLException`**, we need to wrap it in its own **`try ... catch`** block.

```
...
    }catch(SQLException ex){
      System.out.println("\nERROR:----- SQLException -----\n");
      while (ex != null) {
        System.out.println("Message:   " + ex.getMessage());
        System.out.println("SQLState:  " + ex.getSQLState());
        System.out.println("ErrorCode: " + ex.getErrorCode());
        ex = ex.getNextException();
      }
    }catch(Exception e ) {
      e.printStackTrace();
    }finally {
      try {
        if(con != null)
          con.close() ;
      }catch (SQLException ex)  {
        System.out.println("\nERROR:----- SQLException -----\n");
        System.out.println("Message:   " + ex.getMessage());
        System.out.println("SQLState:  " + ex.getSQLState());
        System.out.println("ErrorCode: " + ex.getErrorCode());
      }
    }
  }
...
```

Schema cleanup

What about the (inevitable) need to get rid of tables? The following code snippet demonstrates how to *drop* the tables just created.

```
...
    String fsName = "jdbc/pjtutorial/db2" ;


    Connection con = null ;
```

```
    String dDropSQL = "DROP TABLE digest"  ;
    String mDropSQL = "DROP TABLE messages"  ;
    String aDropSQL = "DROP TABLE authors"  ;
...
      con = ds.getConnection("java", "sun") ;
      Statement stmt = con.createStatement() ;


      stmt.executeUpdate(dDropSQL) ;
      stmt.executeUpdate(mDropSQL) ;
      stmt.executeUpdate(aDropSQL) ;


      System.out.println("Tables Dropped Successfully") ;
...
```

Table population

Once a table has been created, it needs to be populated with the appropriate data. In most applications, the data would either be entered by a user or streamed into the application. For this example, we simply hard-code the relevant data into arrays and loop over them, creating the appropriate SQL DDL statements dynamically. Note that this solution is not very elegant, especially with the **`String`** construction inside the loop. We also might want the database automatically to create the IDs or have a database trigger automatically populate the digest table whenever a new message is written into the messages table.

```
...
    String fsName = "jdbc/pjtutorial/db2" ;


    String baseInsertSQL = "Insert INTO digest VALUES("  ;


    int[] ids = {1, 2, 3, 4, 5} ;
    String[] authors = {"java", "rjb", "java", "bill", "scott"} ;
    String[] titles = { "Hello",
                        "Hello Java",
                        "Hello Robert",
                        "Hello from Bill",
                        "Hello from Scott"} ;
...
      Connection con = ds.getConnection("java", "sun") ;
      Statement stmt = con.createStatement() ;


      for(int i = 0 ; i < ids.length ; i++) {
        stmt.executeUpdate( baseInsertSQL
                                                + ids[i]
                                                + ", '"
                                                + titles[i]
                                                + "', '"
                                                + authors[i]
                                                + "')") ;
      }
...
```

Seeing the results

One of the interesting aspects of database programming is that you often do not get to directly see the results of your work. However, it is easy to write a simple application that displays the newly populated table.

```
...
    // The Data Source name we wish to utilize.

    String fsName = "jdbc/pjtutorial/db2" ;

    String querySQL = "SELECT id, author, title FROM digest" ;
...
      Connection con = ds.getConnection("java", "sun") ;
      Statement stmt = con.createStatement() ;

      ResultSet rs = stmt.executeQuery(querySQL) ;
      ResultSetMetaData rsmd = rs.getMetaData() ;

      for(int i = 1 ; i <= rsmd.getColumnCount() ; i++)
        System.out.print(rsmd.getColumnName(i) + "\t") ;

      System.out.println("\n---------------------------------------") ;

      while(rs.next()) {
        System.out.print(rs.getInt(1) + "\t") ;
        System.out.print(rs.getString(2) + "\t") ;
        System.out.println(rs.getString(3)) ;
      }
...
```

Section 4. Prepared statements

An overview of prepared statements

PreparedStatement objects are different from the normal **Statement** objects used in the last section in two ways. First, they are compiled (prepared) by the JDBC driver or database for faster performance. Second, they accept one or more dynamic input parameters, called **IN** parameters. Together these features make **PreparedStatement** objects suitable for repeated SQL operations where the operations are basically the same with only minor variations, like data loading. For the SQL statement to be prepared before it is used, the SQL must be passed to the JDBC driver when the **PreparedStatement** object is created, not when it is executed.

The **IN** parameters are indicated by **?** placeholders in the SQL **String**. Before the **PreparedStatement** can be successfully executed, the **IN** parameters must be set by calling the **setXXX()** methods on the **PreparedStatement** object, where **XXX** is replaced by the datatype of the parameter being set. Thus, to set the first **IN** parameter to the integer value 100, you would call **setInt(1, 100)**. Likewise, to set the second **IN** parameter to the string value "rjb", you would call **setString(2, "rjb")**. One last point is that a parameter value remains set until it is set to a new value or it is explicitly cleared with a **clearParameters()**. This is important, because a **PreparedStatement** can be executed multiple times; if you are not careful, you can fill your database with garbage.

Prepared insert

The following example inserts data, which is almost identical to the data inserted into the digest table earlier, using a prepared statement. The key difference between the two methods is that the only thing that happens in the loop below is the setting of the three **IN** parameters and the execution of the **PreparedStatement**. This code block could easily be extended to acquire data from a stream and do **Batch** updates to the database.

```
...
    String fsName = "jdbc/pjtutorial/db2" ;

    String insertSQL = "Insert INTO digest VALUES(?, ?, ?)"  ;

    int[] ids = {1, 2, 3, 4, 5} ;
    String[] authors = {"java", "rjb", "java", "bill", "scott"} ;
    String[] titles = { "Prepared Hello",
                        "Prepared Hello Java",
                        "Prepared Hello Robert",
                        "Prepared Hello from Bill",
                        "Prepared Hello from Scott"} ;


...
      Connection con = ds.getConnection("java", "sun") ;

      PreparedStatement pstmt = con.prepareStatement(insertSQL) ;

      for(int i = 0 ; i < ids.length ; i++){
        pstmt.setInt(1, ids[i]) ;
```

```
        pstmt.setString(2, titles[i]) ;
        pstmt.setString(3, authors[i]) ;
        pstmt.executeUpdate() ;
      }
...
```

Prepared query

This example uses a **`PreparedStatement`** to perform a query on the database where the desired message author is selected at run time. You could easily connect this code to a GUI widget, allowing the message board to be searched from a Web application.

```
...
    String fsName = "jdbc/pjtutorial/db2" ;

    String querySQL =
      "SELECT id, author, title FROM digest WHERE author = ?" ;

 ...
      Connection con = ds.getConnection("java", "sun") ;

      PreparedStatement pstmt = con.prepareStatement(querySQL) ;
      pstmt.setString(1, "rjb") ;

      ResultSet rs = pstmt.executeQuery() ;
      ResultSetMetaData rsmd = rs.getMetaData() ;

      for(int i = 1 ; i <= rsmd.getColumnCount() ; i++)
        System.out.print(rsmd.getColumnName(i) + "\t") ;

      System.out.println("\n---------------------------------------") ;

      while(rs.next()) {
        System.out.print(rs.getInt(1) + "\t") ;
        System.out.print(rs.getString(2) + "\t") ;
        System.out.println(rs.getString(3)) ;
      }
...
```

Section 5. Callable statements

An overview of the CallableStatement object

Many databases support functions that execute inside a database. This approach has several benefits, including faster performance and improved security. These functions are called *stored procedures*, and while they are often written in SQL, they can be written in any programming language supported by the database. With the rise in popularity of the Java language, several database vendors, including Oracle and IBM, have enabled the use of the Java language for creating stored procedures, allowing stored procedures to be easily moved between different databases.

Stored procedures can support three types of parameters: **`IN`**, **`OUT`**, and **`INOUT`**, allowing a great deal of flexibility in what a stored procedure can actually do inside the database. Regardless of what language a stored procedure is written in, it is invoked (or called) from a Java application in a standard fashion.

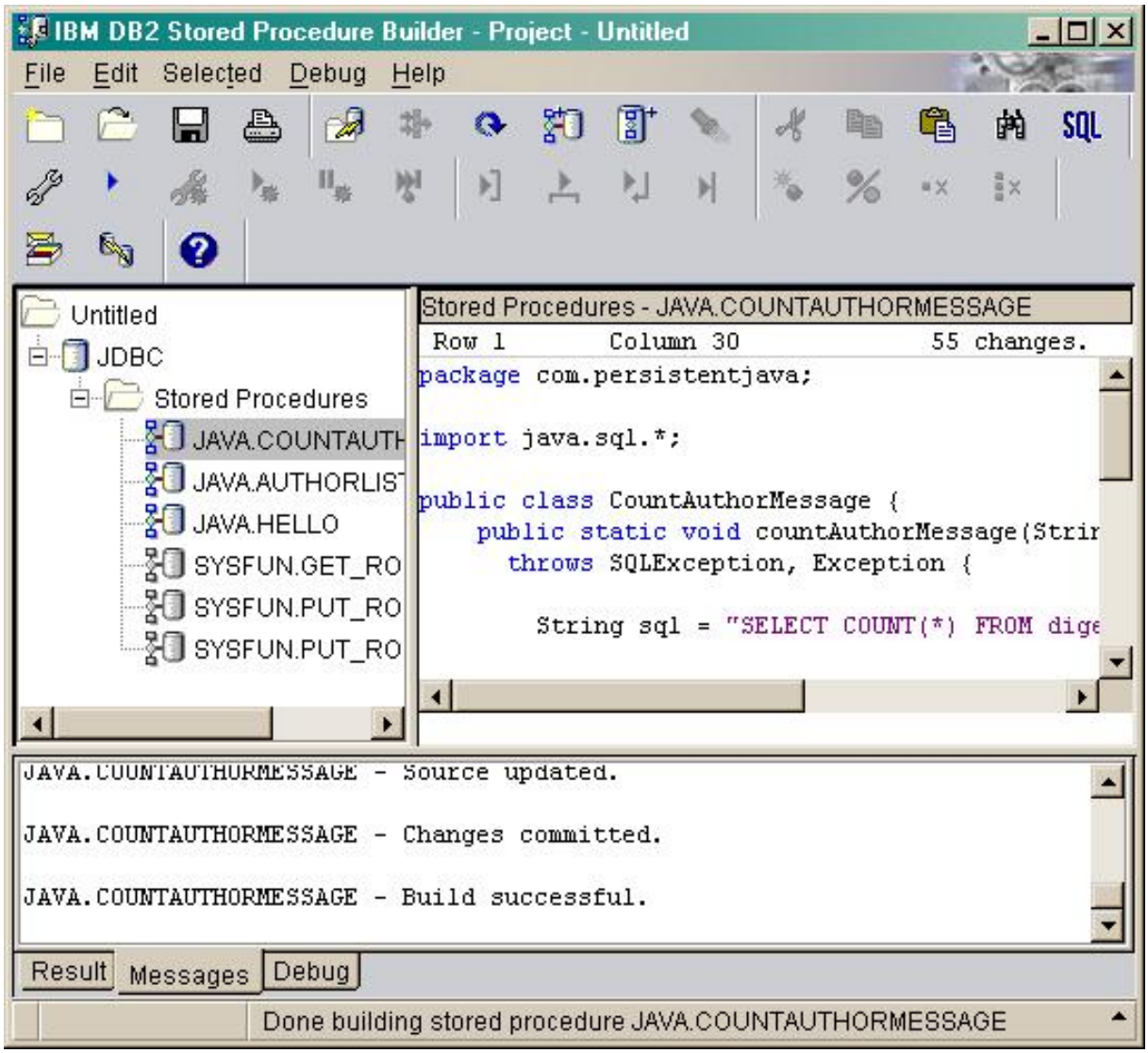
First, you need to create a **`CallableStatement`** object. To identify the stored procedure and the type and number of parameters the procedure expects, three types of invocations are allowed. The following list demonstrates these three types, assuming we are calling a stored procedure named **`AuthorList`**:

* **`{call AuthorList}`** if the procedure takes no parameters
* **`{call AuthorList[(?, ?)]}`** if the procedure takes two parameters
* **`{? = call AuthorList[(?, ?)]}`** if the procedure takes two parameters and returns one

The rest of this section provides several examples of building, inserting, and calling Java stored procedures.

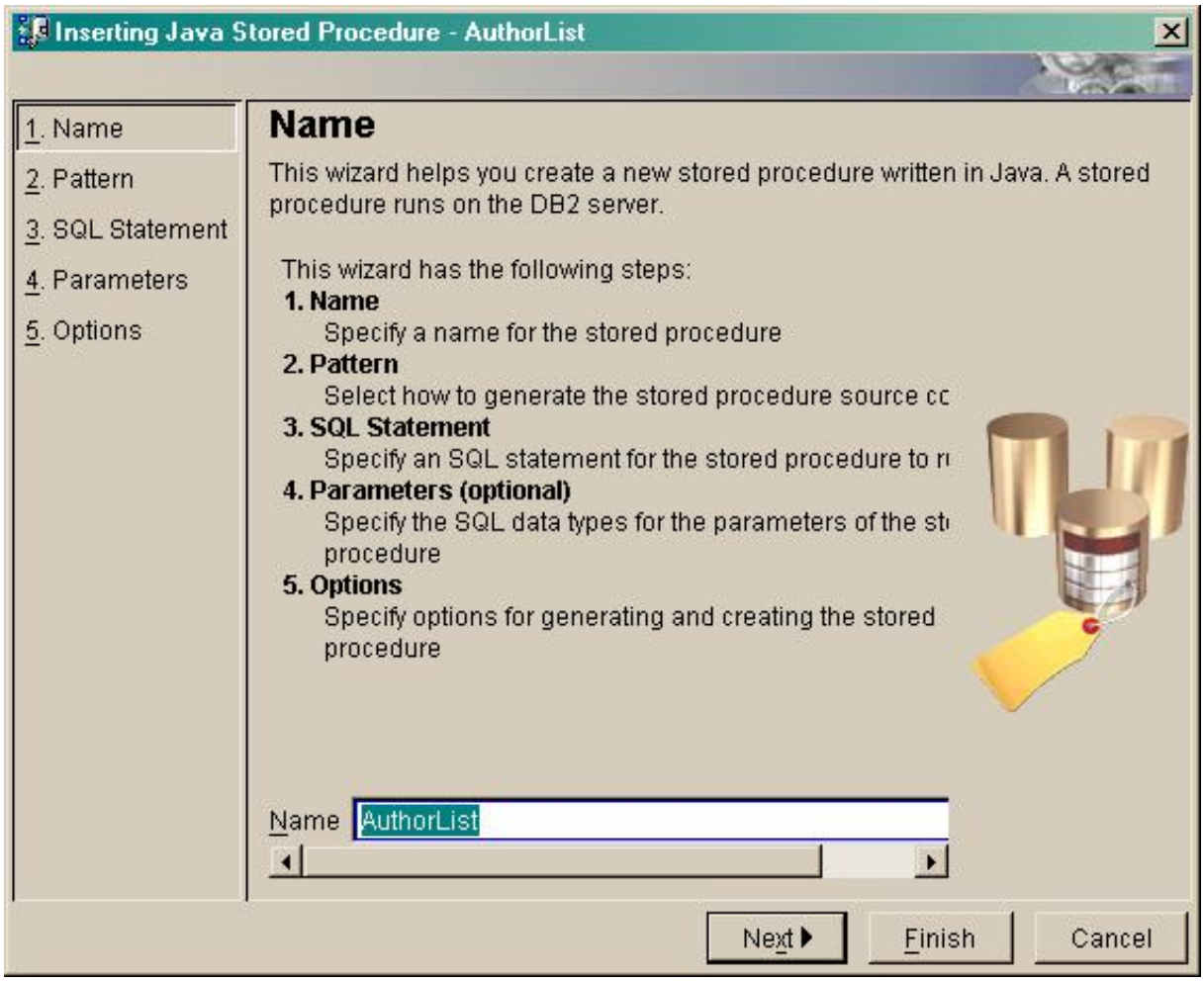
The DB2 Stored Procedure Builder

Creating Java stored procedures in DB2 is greatly simplified by using the Stored Procedure Builder bundled with DB2. Using the Stored Procedure Builder is easy. Start by providing the connection parameters for the database (which, in the context of this tutorial, are the same parameters needed to set up a **`DataSource`**). Once the Builder has made the connection to the database, start creating stored procedures. Note that other database vendors have similar tools to assist the developer in the stored procedure development effort.



The stored procedure build wizard

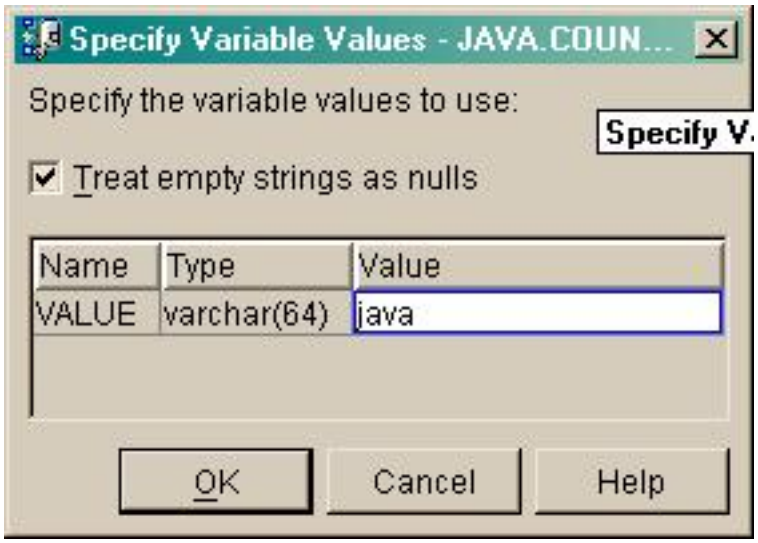
The Stored Procedure Builder lets you insert stored procedures written in either the Java language or SQL into DB2 using either existing files or the stored procedure wizard. By following the prompts provided by the builder wizard, you end up with a skeleton code that has most of what you need to write your stored procedure. You'll still need to provide the heart of the stored procedure. We'll examine two examples later in this section.



Consider these interesting points about the Java code the wizard creates. First, the Java code connects to the database using a default connection. Second, the **IN**, **OUT**, and **INOUT** parameters are automatically generated as part of the method signature.

Calling stored procedures interactively

The DB2 Stored Procedure Builder lets you directly edit the code and rebuild the stored procedure, which in turn greatly simplifies and speeds the development cycle. One other novel feature of the Stored Procedure Builder is its ability to actually call the stored procedure, as demonstrated in the figure below. If **IN** parameters are required, the tool prompts for their values. Likewise, the tool displays the status of all **OUT** parameters.



The AuthorList stored procedure

The following example is a stored procedure that uses one **IN** parameter to generate a list of all messages by that author. Note that the **ResultSet** object is returned as part of an array of **ResultSet** objects, which is necessary because the Java language always passes objects by value. In addition, note that we do not need to handle any error conditions in the code, which is passed onto the calling methods.

```
package com.persistentjava;

import java.sql.*;

public class AuthorList {
    public static void authorList(String value, ResultSet[] rs )
      throws SQLException, Exception {

        String sql =
          "SELECT id, author, title FROM digest WHERE author = ?";

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

        PreparedStatement pstmt = con.prepareStatement(sql);
        pstmt.setString(1, value) ;
        System.err.println(value) ;

        rs[0] = pstmt.executeQuery();

        if (con != null)
          con.close();
    }
}
```

Calling the AuthorList stored procedure

Once a stored procedure is inserted into the database, it can be called from any Java application. The following example calls the **AuthorList** stored procedure printing out the list of messages that have been written by the "java" author.

```
...
    String fsName = "jdbc/pjtutorial/db2" ;

    String callSQL = "{call AuthorList(?)}" ;

...
      Connection con = ds.getConnection("java", "sun") ;

      CallableStatement cstmt = con.prepareCall(callSQL) ;
      cstmt.setString(1, "java") ;

      if(false == cstmt.execute())
        throw new Exception("No ResultSet Returned") ;

      ResultSet rs = cstmt.getResultSet() ;

      ResultSetMetaData rsmd = rs.getMetaData() ;

      for(int i = 1 ; i <= rsmd.getColumnCount() ; i++)
        System.out.print(rsmd.getColumnName(i) + "\t") ;

      System.out.println("\n---------------------------------------") ;

      while(rs.next()) {
        System.out.print(rs.getInt(1) + "\t") ;
        System.out.print(rs.getString(2) + "\t") ;
        System.out.println(rs.getString(3)) ;
      }

      rs.close() ;
      cstmt.close() ;
...
```

The CountAuthorMessage stored procedure

As another example of a Java stored procedure, the **CountAuthorMessage** stored procedure uses an **OUT** parameter to return to the calling application the number of messages that have been written by a given author, which is provided by an **IN** parameter. Notice how this Java stored procedure is nearly identical to a regular JDBC application.

```
package com.persistentjava;

import java.sql.*;

public class CountAuthorMessage {
    public static void countAuthorMessage(String value, int[] count)
      throws SQLException, Exception {
```

```
        String sql = "SELECT COUNT(*) FROM digest WHERE author = ?";

        Connection con = DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement pstmt = con.prepareStatement(sql);
        pstmt.setString(1, value) ;

        ResultSet rs = pstmt.executeQuery();

        rs.next() ;

        count[0] = rs.getInt(1) ;

        if (rs != null)
          rs.close();
        if (pstmt != null)
          pstmt.close();
        if (con != null)
          con.close();
    }
}
```

Calling the CountAuthorMessage stored procedure

The following code snippet calls the **CountAuthorMessage** stored procedure. First we need to create the **CallableStatement** object and set the **IN** parameter. Because this stored procedure uses an **OUT** parameter, we also need to register the **OUT** parameter. To retrieve the **OUT** parameter after executing the stored procedure, call the appropriate **getXXX()** method, where XXX is replaced by the appropriate datatype.

```
...
    String fsName = "jdbc/pjtutorial/db2" ;

    String callSQL = "{call CountAuthorMessage(?, ?)}" ;
...
      Connection con = ds.getConnection("java", "sun") ;

      CallableStatement cstmt = con.prepareCall(callSQL) ;
      cstmt.setString(1, "java") ;
      cstmt.registerOutParameter(2, java.sql.Types.INTEGER) ;

      cstmt.execute() ;
      int count = cstmt.getInt(2) ;

      System.out.println(count + " messages found.") ;

      cstmt.close() ;
...
```

Section 6. Advanced datatypes

An overview of advanced datatypes

The SQL3 standard introduced new, more powerful datatypes into the SQL vernacular. Support for these new datatypes was provided in JDBC 2.0, but JDBC drivers and databases that could provide support for these new datatypes was slow in coming (at least in Internet time). However, the required support has now become rather broad among databases and JDBC drivers, allowing developers to use these new datatypes in their applications. These new datatypes simplify the object-relational impedance mismatch, which occurs because the Java language is object-oriented, and relational databases support the notion of data as tables, or flat rows of columns.

The primary new datatypes include the **Blob** (binary large object), the **Clob** (character large object), the **Array** object, the **REF** (object reference), and the **UDT** (user-defined datatype). Together, these new datatypes allow a database designer to create a richer schema and simplify the handling and persisting of complex data. The rest of this section provides detailed examples showing how to use the **Blob** and the **Clob** objects.

Inserting Blobs

The following code snippet demonstrates how to insert a **Blob** (in this case a picture of "yours truly") into the authors table. Of particular interest here is the use of a stream to supply the image from a file that is inserted directly into the database.

```
...
    String fsName = "jdbc/pjtutorial/db2" ;
    Connection con = null ;

    String insertSQL = "Insert INTO authors VALUES(?, ?)"  ;

...
      con = ds.getConnection("java", "sun") ;

      PreparedStatement pstmt = con.prepareStatement(insertSQL) ;

      File file = new File("C:/images/rjb.jpg") ;
      FileInputStream fis = new FileInputStream(file);

      pstmt.setString(1, "rjb");
      pstmt.setBinaryStream(2, fis, (int)file.length());

      if(1 != pstmt.executeUpdate())
        System.err.println("Incorrect value returned during author insert.") ;

      pstmt.close();
      fis.close();

      System.out.println("BLOB Insert Successful") ;
```

```
  ...
```

Selecting a Blob

The following example demonstrates how to retrieve a **Blob** datatype from the database. In this case, we retrieve the previously inserted image, displaying it in its own **JFrame**. Given that the code is rather lengthy to do this, the example is split over several panels. First, we do a query to select the **Blob** of interest and pull it back to the client (also known as materializing the **Blob**).

```
 ...
public class BlobSelectDB2  extends JPanel {

  public BlobSelectDB2() {

    // The Data Source name we wish to utilize.

    String fsName = "jdbc/pjtutorial/db2" ;
    Connection con = null ;

    String selectSQL = "SELECT photo FROM authors WHERE author = ?"  ;
 ...
      con = ds.getConnection("java", "sun") ;

      PreparedStatement pstmt = con.prepareStatement(selectSQL) ;

      pstmt.setString(1, "rjb");

      ResultSet rs = pstmt.executeQuery() ;

      rs.next() ;

      Blob blob = rs.getBlob("photo") ;

      // Materialize BLOB onto client

      ImageIcon icon = new ImageIcon(blob.getBytes(1, (int)blob.length())) ;

      JLabel lPhoto = new JLabel(icon) ;
      setLayout(new GridLayout(1, 1));

      add(lPhoto) ;

      rs.close();
      pstmt.close();
 ...
```

Displaying a Blob

The rest of the code simply creates a **JFrame** to hold the retrieved image. To make things a little more interesting, this listing also shows how to set the Look-and-Feel of the GUI; simply uncomment the desired Look-and-Feel and recompile to get the new view.

```
public static void main(String args[]){

    try{

      UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel") ;

      // UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel") ;
      // UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel")

    }catch(Exception e){
      System.err.println("\nERROR When Setting the Swing Look and Feel: ");
      System.err.println(e.getMessage());
    }

    JFrame frame = new JFrame("Blob Demo");

    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });

    frame.setContentPane(new BlobSelectDB2()) ;
    frame.pack();
    frame.setVisible(true);
  }
}
```

Seeing the Blob

If you have followed along with all of the examples so far, you should have a **JFrame** like the one to the left displayed after executing BlobSelectDB2.

While rather simple, this example has demonstrated how easy it is to use an advanced datatype, in this case a **Blob** datatype. With a little additional effort, you could build a GUI widget that allows a user to browse a list of photos in a database, or perhaps something even more amazing. Now let's try our hand at a **Clob** object.

Inserting a Clob

The **Clob** example will look very similar to the previous example, which is intentional. In the following code snippet, we fill a **Clob** column in the messages table using a pre-existing message from a file. This example shows how to insert a **Clob** object directly from an ASCII stream.

```
...
    String fsName = "jdbc/pjtutorial/db2" ;
    Connection con = null ;

    String insertSQL = "Insert INTO messages VALUES(?, ?, ?, ?)"  ;
...
     con = ds.getConnection("java", "sun") ;

      PreparedStatement pstmt = con.prepareStatement(insertSQL) ;

      File file = new File("C:/data/rjb.txt") ;
      FileInputStream fis = new FileInputStream(file);

      pstmt.setInt(1, 1);
      pstmt.setString(2, "Hello Java");
      pstmt.setString(3, "rjb");
      pstmt.setAsciiStream(4, fis, (int)file.length());

      if(1 != pstmt.executeUpdate())
        System.err.println("Incorrect value returned during message insert.") ;

      pstmt.close();
      fis.close();

      System.out.println("CLOB Insert Successful") ;
...
```

Selecting a Clob

After we have a **Clob** object in the database, we can retrieve it. The following example is split over several panels due to its length. It shows how to pull the **Clob** object we just inserted out of the database and display it in a **JFrame**.

```
...
public class ClobSelectDB2 extends JPanel {
```

```
  public ClobSelectDB2() {

    // The Data Source name we wish to utilize.

    String fsName = "jdbc/pjtutorial/db2" ;
    Connection con = null ;

    String selectSQL = "SELECT message FROM messages WHERE id = ?"  ;
...
      con = ds.getConnection("java", "sun") ;

      PreparedStatement pstmt = con.prepareStatement(selectSQL) ;

      pstmt.setInt(1, 1);

      ResultSet rs = pstmt.executeQuery() ;

      rs.next() ;

      Clob clob = rs.getClob("message") ;

      // Materialize CLOB onto client

      InputStreamReader in = new InputStreamReader(clob.getAsciiStream()) ;

      JTextArea text = new JTextArea(readString(in)) ;
      setLayout(new GridLayout(1, 1));

      add(text) ;

      rs.close();
      pstmt.close();
...
```

Displaying a Clob

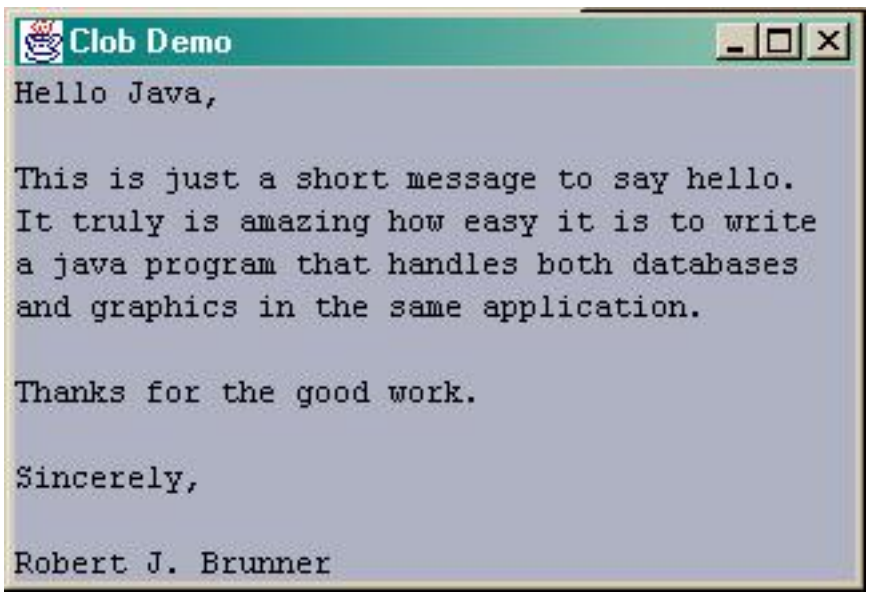
After we've retrieved the **`Clob`**, it's only natural to do something with it. One minor problem: the **`InputStreamReader`** we have obtained for our **`Clob`** object needs to be converted into a string before it can be displayed. (We did not have to do this for the Blob example because the **`IconImage`** object can read an image directly from a stream.) The **`readString()`** method is inspired from recipe 9.5 from Ian Darwin's *Java Cookbook* (O'Reilly and Associates, 2001).

```
...
  public static String readString(InputStreamReader in) throws IOException {
    StringBuffer buf = new StringBuffer() ;
    int n ;
    char[] b = new char[1024] ;
```

```
    while((n = in.read(b)) > 0)
      buf.append(b, 0, n) ;

    return buf.toString() ;
  }

  public static void main(String args[]){
...
    JFrame frame = new JFrame("Clob Demo");

    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });

    frame.setContentPane(new ClobSelectDB2()) ;
    frame.pack();
    frame.setVisible(true);
  }
}
```

Seeing the Clob

If you're following along with the examples, you should see the **JFrame** below displayed after executing ClobSelectDB2. Again, this rather simple example demonstrates how easy it is to use advanced SQL datatypes from Java applications.



Section 7. Summary

Advanced Java database operations

Hopefully this tutorial clearly explained how to perform advanced database operations from Java applications. Although there is a great deal of information available online, it can be hard to digest and is often confusing. To remedy this situation, this tutorial presented an entirely self-contained introduction to building a simple database application that explained:

* How to design a database, including creating a database schema
* How to create and use a **`PreparedStatement`**
* How to create and use a **`CallableStatement`**
* How to use some of the advanced datatypes that were introduced in JDBC 2.0

Now all that remains is for you to go out and build your own, even better, Java database application.

Resources

Download the source code used in this tutorial, brunner2-code.zip.

JDBC information

* Visit the official *JDBC home page* for the JDBC 2.0 and 3.0 specifications and other information.

* *JDBC API Tutorial and Reference, Second Edition* (Addison-Wesley, 1999) by White, Fisher, Cattell, Hamilton, and Hapner is *the* reference for JDBC developers.

* "*Managing database connections with JDBC*" (developerWorks, November 2001), another JDBC tutorial by Robert Brunner, provides an introduction to the different concepts involved in establishing and managing a database connection from within a Java application using JDBC.

* Lennart Jorelid shows you how to use JDBC for *industrial-strength performance* .

* "*What's new in JDBC 3.0*" (develperWorks, July 2001) by Josh Heidebrecht provides an overview of the new features and enhancements in the new spec.

* "*An easy JDBC wrapper*" (developerWorks, August 2001) by Greg Travis describes a simple wrapper library that makes basic database usage a snap.

* The *Java application development with DB2* Web site provides an important collection of useful DB2 and Java links.

* The *Java Naming and Directory Interface* (JNDI) Web site provides a wealth of information regarding naming services, including an excellent JNDI tutorial. To use the JNDI examples, you will need to download the filesystem context provider, which is available from the site.

JDBC driver information

* To find a JDBC driver for a particular database, visit Sun's *searchable database of JDBC drivers*.
* *Merant* is a third-party vendor that provides JDBC drivers for a range of databases.
* *i-net software* is another third-party vendor for JDBC drivers.
* *SourceForge* offers an open source JDBC driver for the MySQL database.
* The *Center for Imaginary Environments* offers an open source JDBC driver for the mSQL database.

Database resources

* Much of the example code in this tutorial was developed using *DB2 Universal Database*. If you're using this platform or wish to learn more about it from a technical perspective, visit the *DB2 Developer Domain*, a centralized technical resource for the DB2 developer community.
* The following links provide further reading on SQL:
    * The *SQL Reference Page* provides numerous links to general SQL topics and information, SQL syntax information, programming resources, the many different "blends" of SQL, SQL tools, and generic DBMS information.
    * *SQL for Web Nerds* contains detailed discussions on data modeling, construction queries, triggers, views, trees, and data warehousing.

Feedback

We welcome your feedback on this tutorial, and look forward to hearing from you. Additionally, you are welcome to contact the author, Robert Brunner, directly at rjbrunner@pacbell.net.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the

Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.