

CHAPTER 18

Data Access Using JDBC

About this chapter

This chapter describes how to use JDBC to access data.

JDBC can be used both from client applications and inside the database. Java classes using JDBC provide a more powerful alternative to SQL stored procedures for incorporating programming logic in the database.

Contents

Topic	Page
JDBC overview	504
Establishing JDBC connections	507
Using JDBC to access data	515
Using the Sybase jConnect JDBC driver	523
Creating distributed applications	527

JDBC overview

JDBC provides a SQL interface for Java applications: if you want to access relational data from Java, you do so using JDBC calls.

This chapter is not a thorough guide to the JDBC database interface. Instead, it provides some simple examples to introduce JDBC and to illustrate how it can be used inside and outside the server. It provides more details on the server-side use of JDBC, running inside the database server.

☞ The examples illustrate the distinctive features of using JDBC in Adaptive Server Anywhere. For more information about JDBC programming, see any JDBC programming book.

JDBC and Adaptive Server Anywhere

You can use JDBC with Adaptive Server Anywhere in the following ways:

- ◆ **JDBC on the client** Java client applications can make JDBC calls to Adaptive Server Anywhere. The connection takes place through the Sybase jConnect JDBC driver.

In this chapter, the phrase **client application** applies both to applications running on a user's machine and to logic running on a middle-tier application server.

- ◆ **JDBC in the server** Java classes installed into a database can make JDBC calls to access and modify data in the database, using an internal JDBC driver.

The focus in this chapter is on server-side JDBC.

JDBC resources

- ◆ **Required software** You need the Sybase jConnect driver and TCP/IP in order to use JDBC from an external application.

The Sybase jConnect driver may already be available, depending on your installation of Adaptive Server Anywhere. For more information about the jConnect driver and its location, see "The jConnect driver files" on page 523.

- ◆ **Example source code** Source code for the examples in this chapter can be found in the file *JDBCExamples.java* in the *jxmp* subdirectory under your Adaptive Server Anywhere installation directory.

☞ For instructions on how to install the Java examples, including the *JDBCExamples* class, see "Installing the Java examples" on page 466.

JDBC program structure

The following sequence of events is typical of a JDBC application:

- 1 **Create a Connection object** A `Connection` object is created by calling `getConnection` class method of the `DriverManager` class. This establishes a connection with a database.
- 2 **Generate a Statement object** The `Connection` object is used to generate a `Statement` object.
- 3 **Pass a SQL statement** The `Statement` object is passed a SQL statement that is executed within the database environment. If the statement is a query, this action causes a `ResultSet` object to be returned.

The `ResultSet` object contains the data returned from the SQL statement, but exposes it one row at a time (similar to the way a cursor works).

- 4 **Loop over the rows of the result set** The `next` method of the `ResultSet` object performs two actions:
 - ◆ The current row (the row in the result set which is being exposed through the `ResultSet` object) is advanced one row.
 - ◆ A Boolean value (true/false) is returned to indicate whether there is, in fact, a row to advance to.
- 5 **For each row, retrieve the values** Values are retrieved for each column in the `ResultSet` object by identifying either the name or position of the column. The `getDate` method is one method used to get the value from a column on the current row.

Java objects can use JDBC objects to interact with a database and get data for their own use, such as to manipulate or for use in other queries.

Differences between client- and server-side JDBC

The difference between JDBC on the client and in the database server is in establishing a connection with the database environment.

- ◆ **Client side** In client-side JDBC, establishing a connection requires the Sybase jConnect JDBC driver. The connection is established by passing arguments to the `DriverManager.getConnection` method. The database environment is an external application from the perspective of the client application.

jConnect required

Depending on the package in which you received Adaptive Server Anywhere, Sybase jConnect may or may not be included. You must have jConnect in order to use JDBC from external applications. You can use internal JDBC without jConnect.

- ◆ **Server-side** When JDBC is used within the database server, a connection already exists. A value of **jdbc:default:connection** is passed to `DriverManager.getConnection`, which provides the JDBC application with the ability to work within the current user connection. This is a quick, efficient and safe operation because the client application has already passed the database security in order to establish the connection: the user ID and password have been provided and do not need to be provided again.

You can write JDBC classes in such a way that they can be run both at the client and at the server by employing a single conditional statement for constructing the URL. An external connection requires the machine name and port number, while the internal connection requires **jdbc:default:connection**.

Server-side JDBC
version

The internal JDBC driver supports JDBC version 1.1.

Establishing JDBC connections

This section presents classes that establish a JDBC database connection from a Java application.

Connecting from a JDBC client application

If you wish to access database system tables (database metadata) from a JDBC application, you must add a set of jConnect system objects to your database. If you do not need to access the system tables, you do not need to add these objects.

☞ For information about adding the jConnect system objects to a database, see "Using the Sybase jConnect JDBC driver" on page 523.

The following complete Java application is a command-line application that connects to a running database, prints a set of information to your command line, and terminates.

Establishing a connection is the first step any JDBC application must take when working with database data.

☞ This example illustrates an external connection, which is a regular client/server connection. For information on how to create an internal connection, from Java classes running inside the database server, see "Establishing a connection from a server-side JDBC class" on page 511.

External connection example code

The following is the source code for the methods used to make a connection. The source code can be found in the `main` method and the `ASACConnect` method of the file `JDBCExamples.java` in the `jxmp` directory under your Adaptive Server Anywhere installation directory:

```
// Import the necessary classes
import java.sql.*;           // JDBC
import com.sybase.jdbc.*;   // Sybase jConnect
import java.util.Properties; // Properties
import sybase.sql.*;        // Sybase utilities
import asademo.*;           // Example classes

private static Connection conn;

public static void main(String args[]) {

    conn = null;
    String machineName;
```

```
if ( args.length != 1 ) {
    machineName = "localhost";
} else {
    machineName = new String( args[0] );
}

ASAConnect( "dba", "sql", machineName );
if( conn!=null ) {
    System.out.println( "Connection successful" );
}else{
    System.out.println( "Connection failed" );
}

try{
    serializeVariable();
    serializeColumn();
    serializeColumnCastClass();
}
catch( Exception e ) {
    System.out.println( "Error: " + e.getMessage() );
    e.printStackTrace();
}
}

private static void ASAConnect( String UserID,
                                String Password,
                                String Machinename ) {
    // uses global Connection variable

    String _coninfo = new String( Machinename );

    Properties _props = new Properties();
    _props.put("user", UserID );
    _props.put("password", Password );

    // Load the Sybase Driver
    try {

Class.forName("com.sybase.jdbc.SybDriver").newInstance()
;

        StringBuffer temp = new StringBuffer();
        // Use the Sybase jConnect driver...
        temp.append("jdbc:sybase:Tds:");
        // to connect to the supplied machine name...
        temp.append( _coninfo );
        // on the default port number for ASA...
        temp.append(":2638");
        // and connect.
        System.out.println(temp.toString());
    }
}
```

```

        conn = DriverManager.getConnection(
temp.toString() , _props );
    }
    catch ( Exception e ) {
        System.out.println("Error: " + e.getMessage());
        e.printStackTrace();
    }
}

```

How the external connection example works

	The external connection example is a Java command-line application.
Importing packages	<p>The application requires several libraries, which are imported in the first lines of <i>JDBCExamples.java</i>:</p> <ul style="list-style-type: none"> ◆ The Sun Microsystems JDBC classes are contained in the <code>java.sql</code> package, and are required for all JDBC applications. It is contained in the <i>classes.zip</i> file in your Java subdirectory. ◆ The Sybase jConnect JDBC driver is imported from <code>com.sybase.jdbc</code>. This is required for all applications that connect using jConnect. It is contained in the <i>jdbcdrv.zip</i> file in your Java subdirectory. ◆ The application uses a property list. The <code>java.util.Properties</code> class is required to handle property lists. It is contained in the <i>classes.zip</i> file in your <i>Java</i> subdirectory. ◆ The <code>sybase.sql</code> package contains utilities used for serialization. It is contained in the <i>asajdbc.zip</i> file in your <i>Java</i> subdirectory. ◆ The <code>asademo</code> package contains example classes used in some examples. It is contained in the <i>asademo.jar</i> file in your <i>jxmp</i> subdirectory.
The main method	<p>Each Java application requires a class with a method named <code>main</code>, which is the method invoked when the program is started. In this simple example, <code>JDBCExamples.main</code> is the only method in the application.</p> <p>The <code>JDBCExamples.main</code> method carries out the following tasks:</p> <ol style="list-style-type: none"> 1 Processes the command-line argument. If a machine name is supplied, then this is used. By default, the machine name is set to <i>localhost</i>, which is appropriate for the personal database server. 2 Calls the <code>ASAConnect</code> method to establish a connection. 3 Executes several methods that scroll data to your command line.

The ASAConnect method

The `JDBCExamples.ASAConnet` method carries out the following tasks:

- 1 Connects to the default running database using Sybase jConnect.
 - ◆ `Class.forName` loads jConnect. Using the `newInstance` method works around issues in some browsers.
 - ◆ The `StringBuffer` statements build up a connection string from the literal strings and the supplied machine name provided on the command line.
 - ◆ `DriverManager.getConnection` establishes a connection using the connection string.
- 2 Returns control to the calling method.

Running the external connection example

This section describes how to run the external connection example

❖ **To create and execute the external connection example application:**

- 1 From a system command prompt, change to the Adaptive Server Anywhere installation directory. Then change to the *jxmp* subdirectory
- 2 Ensure that your CLASSPATH environment variable is set to include the current directory (.) and the zip files that are imported. For example, from a command prompt (the following should be entered all on one line):

```
set
classpath=..\java\jdbcdrv.zip;..\java\asajdbc.zip;
asademo.jar
```

The default zip file name for Java is *classes.zip*. For classes in any file named *classes.zip*, you only need the directory name in the CLASSPATH variable, not the zip-file name itself. For classes in files with other names, you must supply the zip file name.

You need the current directory in the CLASSPATH in order to run the example.

- 3 Ensure that the database is loaded onto a database server running TCP/IP. You can start such as server on your local machine using the following command (from the *jxmp* subdirectory):

```
start dbeng6 -c 8M ..\asademo
```

- 4 Run the example by entering the following at the command prompt:


```
java JDBCExamples
```

If you wish to try this against a server running on another machine, you must enter the correct name of that machine. The default is **localhost**, which is as an alias for the current machine name.

- 5 Confirm that a list of people and products is displayed at your command prompt.

If the attempt to connect fails, an error message is displayed instead. Confirm that you have executed all the steps as required. The most common mistake is to have an incorrect CLASSPATH, which results in a failure to locate a class.

☞ For more information about using jConnect, see "Using the Sybase jConnect JDBC driver" on page 523 and the online documentation for jConnect.

Establishing a connection from a server-side JDBC class

SQL statements in JDBC are built using the `createStatement` method of a `Connection` object. Even classes running inside the server need to establish a connection in order to create a `Connection` object.

Establishing a connection from a server-side JDBC class is more straightforward than establishing an external connection. Because the server-side class is executed by a user already connected, the class simply uses the current connection.

Server-side connection example code

The following is the source code for the example. The source code can be found in the `InternalConnect` method of `JDBCExamples.java` in the `jxmp` directory under your Adaptive Server Anywhere installation directory:

```
public static void InternalConnect() {
    try {
        conn =
        DriverManager.getConnection("jdbc:default:connection");
        System.out.println("Hello World");
    }
    catch ( Exception e ) {
        System.out.println("Error: " + e.getMessage());
        e.printStackTrace();
    }
}
```

How the server-side connection example works

In this simple example, `InternalConnect ()` is the only method used in the application.

The application requires only one of the libraries (JDBC) imported in the first line of the `JDBCExamples.java` class. The others are required for external connections. The JDBC classes are contained in the package named `java.sql`.

The `InternalConnect ()` method carries out the following tasks:

- 1 Connects to the default running database using the current connection:
 - ◆ `DriverManager.getConnection` establishes a connection using a connection string of `jdbc:default:connection`.
- 2 Prints **Hello World** to the current standard output, which is the server window. `System.out.println` carries out the printing.
- 3 If there is an error in the attempt to connect, an error message is printed to the server window, together with the place where the error occurred.

The `try` and `catch` instructions provide the framework for the error handling.
- 4 The class terminates.

Running the server-side connection example

This section describes how to run the server-side connection example.

❖ To create and execute the internal connection example application:

- 1 If you have not already done so, compile the `JDBCExamples.java` file. If you are using the JDK, you can do the following in the `jxmp` directory from a command prompt:

```
javac JDBCExamples.java
```

- 2 Start a database server using the sample database. You can start such as server on your local machine using the following command (from the `jxmp` subdirectory):

```
start dbeng -c 8M ..\asademo
```

The TCP/IP network protocol is not required in this case, as `jConnect` is not being used. However, you must have at least 8 Mb of cache available to use Java classes in the database.

- 3 Install the class into the sample database. Once connected to the sample database, you can do this from Interactive SQL using the following command:

```
INSTALL JAVA NEW
FROM FILE 'path\jxmp\JDBCEexamples.class'
```

where *path* is the path to your installation directory.

You can also install the class using Sybase Central. While connected to the sample database, open the Java Objects folder and double click Add Class. Then follow the instructions in the wizard.

- 4 You can now call the **InternalConnect** method of this class just as you would an stored procedure:

```
CALL JDBCEexamples>>InternalConnect()
```

The first time a Java class is called in a session, the internal Java virtual machine must be loaded. This can take a few seconds.

- 5 Confirm that the message **Hello World** is printed on the server screen.

Notes on JDBC connections

- ◆ **Autocommit behavior** The JDBC specification requires that, by default, a COMMIT is performed after each data modification statement. Currently, the server-side JDBC behavior is not to commit. You can control this behavior using a statement such as the following:

```
conn.setAutoCommit( false );
```

where **conn** is the current connection object.

- ◆ **Connection defaults** From server-side JDBC, only the first call to **getConnection("jdbc:default:connection")** creates a new connection with the default values. Subsequent calls return a wrapper of the current connection with the all connection properties unchanged. If you set AutoCommit to OFF in your initial connection, any subsequent **getConnection** calls within the same Java code return a connection with AutoCommit set to OFF.

You may wish to ensure that connection properties are reset to their default values when a connection is closed, so that subsequent connections are obtained with standard JDBC values. The following type of code achieves this:

```
Connection conn = DriverManager.getConnection("");
boolean oldAutoCommit = conn.getAutoCommit();
try {
    // do code here
```

```
    }  
    finally {  
        conn.setAutoCommit( oldAutoCommit );  
    }  
}
```

This discussion applies not only to `AutoCommit`, but also to other connection properties such as `TransactionIsolation` and `isReadOnly`.

Using JDBC to access data

Java applications that hold some or all classes in the database have significant advantages over traditional SQL stored procedures.

At an introductory level, however, it may be helpful to use the parallels with SQL stored procedures to demonstrate the capabilities of JDBC. In the following examples, we write Java classes that insert a row into the **Department** table.

As with other interfaces, SQL statements in JDBC can be either **static** or **dynamic**. Static SQL statements are constructed in the Java application, and sent to the database. The database server parses the statement, and selects an execution plan, and executes the statement. Together, parsing and selecting an execution plan are referred to as **preparing** the statement.

If a similar statement has to be executed many times (many inserts into one table, for example), there can be significant overhead in static SQL because the preparation step has to be executed each time.

In contrast, a dynamic SQL statement contains placeholders. The statement is prepared once, using these placeholders, and the statement can be executed many times without the addition expense of preparing.

In this section we use static SQL. Dynamic SQL is discussed in a later section.

Preparing for the examples

This section describes how to prepare for the examples in the remainder of the chapter.

Sample code

The code fragments in this section are taken from the complete class *JDBCExamples.java*, included in the *jxmp* directory under your installation directory.

❖ To install the JDBCExamples class:

- 1 If you have not already done so, install the *JDBCExamples.class* file into the sample database. Once connected to the sample database from Interactive SQL, you can use the following command:

```
INSTALL JAVA NEW
FROM FILE 'path\jxmp\JDBCExamples.class'
```

where *path* is the path to your installation directory.

You can also install the class using Sybase Central. While connected to the sample database, open the Java Objects folder and double click Add Class. Then follow the instructions in the wizard.

Inserts, updates, and deletes using JDBC

The **Statement** object is used for executing static SQL statements. You execute SQL statements such as INSERT, UPDATE, and DELETE, which do not return result sets, using the **executeUpdate** method of the **Statement** object. Statements such as CREATE TABLE and other data definition statements can also be executed using **executeUpdate**.

The following code fragment illustrates how INSERT statements are carried out within JDBC. It uses an internal connection held in the Connection object named **conn**. The code for inserting values from an external application using JDBC would need to use a different connection, but otherwise would be unchanged.

```
public static void InsertFixed() {
    // returns current connection
    conn =
    DriverManager.getConnection("jdbc:default:connection");
    // Disable autocommit
    conn.setAutoCommit( false );

    Statement stmt = conn.createStatement();

    Integer IRows = new Integer( stmt.executeUpdate
        ("INSERT INTO Department (dept_id, dept_name )"
        + "VALUES (201, 'Eastern Sales')"
        ) );
    // Print the number of rows updated
    System.out.println(IRows.toString() + " row
    inserted" );
}
```

Source code available

This code fragment is part of the **InsertFixed** method of the **JDBCExamples** class, included in the *jxmp* subdirectory of your installation directory.


Notes

- ◆ The **setAutoCommit** method turns off the AutoCommit behavior, so that changes are only committed if an explicit COMMIT instruction is executed.

- ◆ The `executeUpdate` method returns an integer, which reflects the number of rows affected by the operation. In this case, a successful INSERT would return a value of one (1).
- ◆ The integer return type is converted to an `Integer` object. The `Integer` class is a wrapper around the basic `int` data type, providing some useful methods such as `toString()`.
- ◆ The `Integer` `IRows` is converted to a string in order to be printed. The output goes to the server window.

❖ **To run the JDBC Insert example:**

- 1 Using Interactive SQL, connect to the sample database as user ID `dba`.
- 2 Ensure the `JDBCExamples` class has been installed. It is installed together with the other Java examples classes.

 For instructions on installing the Java examples classes, see "Installing the Java examples" on page 466.

- 3 Call the method as follows:

```
CALL JDBCExamples>>InsertFixed()
```

- 4 Confirm that a row has been added to the **department** table.

```
SELECT *
FROM department
```

The row with ID 201 is not committed. You can execute a `ROLLBACK` statement to remove the row.

In this example, you have seen how to create a very simple JDBC class. In subsequent examples, we shall expand on this.

Passing arguments to Java methods

We can expand the `InsertFixed` method to illustrate how arguments are passed to Java methods.

The following method uses arguments passed in the call to the method as the values to be inserted:

```
public static void InsertArguments(
    String id, String name) {
    try {
        conn = DriverManager.getConnection(
            "jdbc:default:connection" );

        String sqlStr = "INSERT INTO Department "
```

```
+ " ( dept_id, dept_name )"
+ " VALUES (" + id + ", ' " + name + "')";

// Execute the statement
Statement stmt = conn.createStatement();
Integer IRows = new Integer( stmt.executeUpdate(
sqlStr.toString() ) );

// Print the number of rows updated
System.out.println(IRows.toString() + " row
inserted" );
}
catch ( Exception e ) {
System.out.println("Error: " + e.getMessage());
e.printStackTrace();
}
}
```

Notes

- ◆ The two arguments are the department id (an integer) and the department name (a string). Here, both arguments are passed to the method as strings, because they are used as part of the SQL statement string.
- ◆ The INSERT is a static statement: it takes no parameters other than the SQL itself.
- ◆ If you supply the wrong number or type of arguments, you receive the Procedure Not Found error.

❖ To use a Java method with arguments:

- 1 If you have not already installed the *JDBCExamples.class* file into the sample database, do so.
- 2 Connect to the sample database from Interactive SQL, and enter the following command:

```
call JDBCExamples>>InsertArguments( '203', 'Northern
Sales' )
```

- 3 Verify that an additional row has been added to the Department table:

```
SELECT *
FROM Department
```

- 4 Roll back the changes to leave the database unchanged:

```
ROLLBACK
```


Queries using JDBC

The **Statement** object is used for executing static queries, as well as statements that do not return result sets. For queries, you use the **executeQuery** method of the **Statement** object. This returns the result set in a **ResultSet** object.

The following code fragment illustrates how queries can be handled within JDBC. The code fragment places the total inventory value for a product into a variable named **inventory**. The product name is held in the **String** variable **prodname**. This example is available as the **Query** method of the **JDBCExamples** class.

The example assumes an internal or external connection has been obtained and is held in the **Connection** object named **conn**. It also assumes a variable

```
public static void Query () {
    int max_price = 0;
    try{
        conn = DriverManager.getConnection(
            "jdbc:default:connection" );

        // Build the query
        String sqlStr = "SELECT id, unit_price "
        + "FROM product" ;

        // Execute the statement
        Statement stmt = conn.createStatement();
        ResultSet result = stmt.executeQuery( sqlStr );

        while( result.next() ) {
            int price = result.getInt(2);
            System.out.println( "Price is " + price );
            if( price > max_price ) {
                max_price = price ;
            }
        }
    }
    catch( Exception e ) {
        System.out.println("Error: " + e.getMessage());
        e.printStackTrace();
    }
    return max_price;
}
```

Running the example

Once you have installed the **JDBCExamples** class into the sample database, you can execute this method using the following statement in Interactive SQL:

```
select JDBCExamples>>Query()
```

Notes

- ◆ The query selects the quantity and unit price for all products named **prodname**. These results are returned into the **ResultSet** object named **result**.
- ◆ There is a loop over each of the rows of the result set. The loop uses the **next** method.
- ◆ For each row, the value of the each column is retrieved into an integer variable using the **getInt** method. **ResultSet** also has methods for other data types, such as **getString**, **getDate**, and **getBinaryString**.

The argument for the **getInt** method is an index number for the column, starting from 1

Data type conversion from SQL to Java is carried out according to the information in "SQL-to-Java data type conversion" on page 257 of the book *Adaptive Server Anywhere Reference Manual*.
- ◆ Adaptive Server Anywhere supports bidirectional scrolling cursors. However, JDBC provides only the **next** method, which corresponds to scrolling forward through the result set.
- ◆ The method returns the value of **max_price** to the calling environment, and Interactive SQL displays it in the Data window.

Using prepared statements for more efficient access

If you use the **Statement** interface, each statement you send to the database must be parsed, an access plan must be generated, and the statement must be executed. The steps prior to actual execution are called **preparing** the statement.

You can achieve performance benefits if you use the **PreparedStatement** interface. This allows you to prepare a statement using placeholders, and then assign values to the placeholders when executing the statement.

Using prepared statements is particularly useful when carrying out many similar actions, such as inserting many rows.

☞ For a general introduction to prepared statements, see "Preparing statements" on page 202.

Example

The example actually inserts a single row, which is not a good use of prepared statements. Nevertheless, it illustrates how to use the **PreparedStatement** interface.

The following method of the **JDBCExamples** class carries out a prepared statement:

```
public static void JInsertPrepared(int id, String name)
try {
    conn = DriverManager.getConnection(
        "jdbc:default:connection");

    // Build the INSERT statement
    // ? is a placeholder character
    String sqlStr = "INSERT INTO Department "
+ "( dept_id, dept_name ) "
+ "VALUES ( ? , ? )" ;

    // Prepare the statement
    PreparedStatement stmt = conn.prepareStatement(
sqlStr );

    stmt.setInt(1, id);
    stmt.setString(2, name );
    Integer IRows = new Integer(
        stmt.executeUpdate() );

    // Print the number of rows updated
    System.out.println(IRows.toString() + " row
inserted" );
}
catch ( Exception e ) {
    System.out.println("Error: " + e.getMessage());
    e.printStackTrace();
}
}
```

Running the
example


Once you have installed the **JDBCExamples** class into the sample database, you can execute this example by entering the following statement:

```
call JDBCExamples>>InsertPrepared(
    202, 'Eastern Sales' )
```

The string argument is enclosed in single quotes, which is appropriate for SQL. If you invoked this method from a Java application you would use double quotes to delimit the string.

Inserting and retrieving objects

As an interface to relational databases, JDBC is designed to retrieve and manipulate traditional SQL data types. Adaptive Server Anywhere also provides abstract data types in the form of Java classes. The way you access these Java classes using JDBC depends on whether you are inserting or retrieving the objects.

 For more information on getting and setting entire objects, see "Creating distributed applications" on page 527.

Retrieving objects

You can retrieve objects and their fields and methods in the following ways:

- ◆ **Accessing methods and fields** Java methods and fields can be included in the select-list of a query. A method or field then appears as a column in the result set, and can be accessed using one of the standard `ResultSet` methods, such as `getInt`, or `getString`.
- ◆ **Retrieving an object** If you include a column with a Java class data type in a query select list, you can use the `ResultSet getObject` method to retrieve the object into a Java class. You can then access the methods and fields of that object within the Java class.

Inserting objects

From a server-side Java class, you can use the JDBC `setObject` method to insert an object into a column with Java class data type.

Inserting objects can be carried out using a prepared statement. For example, the following code fragment inserts an object of type `MyJavaClass` into a column of table T:

```
java.sql.PreparedStatement ps =
    conn.prepareStatement("insert T values( ? )" );
ps.setObject( 1, new MyJavaClass() );
ps.executeUpdate();
```

An alternative is to set up a SQL variable that holds the object and then to insert the SQL variable into the table.


Miscellaneous JDBC notes

- ◆ **Access permissions** Like all Java classes in the database, classes containing JDBC statements can be accessed by any user. There is no equivalent of the GRANT EXECUTE statement that grants permission to execute procedures, and there is no need to qualify the name of a class with the name of its owner.
- ◆ **Execution permissions** Java classes are executed with the permissions of the connection executing them. This behavior is different to that of stored procedures, which execute with the permissions of the owner.

Using the Sybase jConnect JDBC driver

If you wish to use JDBC from a client application or applet, you must have Sybase jConnect to connect to Adaptive Server Anywhere databases.

Depending on the package in which you received Adaptive Server Anywhere, Sybase jConnect may or may not be included. You must have jConnect in order to use JDBC from client applications. You can use server-side JDBC without jConnect.

 For a full description of jConnect and its use with Adaptive Server Anywhere, see the jConnect documentation available in the online Books or from the jConnect Web site.

Versions of jConnect supplied with Adaptive Server Anywhere

Adaptive Server Anywhere provides the following versions of the Sybase jConnect JDBC driver:

- ◆ **Full version** If you choose to install jConnect, a jConnect subdirectory is added to your Adaptive Server Anywhere installation. This holds a directory tree with all jConnect files.
- ◆ **Zip file** The Remote Data Access features, and the Java debugger, both use jConnect to connect to the database. A zip file of the basic jConnect classes is provided to enable jConnect use even without the full development version of the driver.

The jConnect driver files

The Sybase jConnect driver is installed into a set of directories under the *jConnect* subdirectory of your Adaptive Server Anywhere installation. If you have not installed jConnect, you can use the *jdbcdrv.zip* file installed into the Java subdirectory.

Classpath setting for jConnect

For your application to use jConnect, the jConnect classes must be in your CLASSPATH environment variable at compile time and run time, so that the Java compiler and Java runtime can locate the necessary files.

For example, the following command adds the jConnect driver class path to an existing CLASSPATH environment variable where *path* is your Adaptive Server Anywhere installation directory.

```
set classpath=%classpath%;path\jConnect\classes
```

The following alternative command adds the *jdbcdrv.zip* file to your CLASSPATH.

Importing the jConnect classes

```
set classpath=%classpath%;path\java\jdbcdrv.zip
```

The classes in jConnect are all in the `com.sybase` package. The client application needs to access classes in `com.sybase.jdbc`. For your application to use jConnect, you must import these classes at the beginning of each source file:

```
import com.sybase.jdbc.*
```

Installing jConnect system objects into a database

If you wish to use jConnect to access system table information (database metadata), you must add the jConnect system objects to your database.

By default, the jConnect system objects are added to a database for any database created using Version 6, and to any database upgraded to Version 6.

If you do not add the jConnect objects to the database when creating or upgrading, you can add them at a later time.

You can install the jConnect system objects from Sybase Central or from Interactive SQL.

❖ To add the jConnect system objects to a Version 6 database from Sybase Central:

- 1 Connect to the database from Sybase Central as a user with DBA authority.
- 2 Open the Java Objects folder.
- 3 Double-click Re-Install jConnect Meta-data Support and follow the instructions in the wizard.

❖ To add the jConnect system objects to a Version 6 database from Interactive SQL:

- ◆ Connect to the database from Interactive SQL as a user with DBA authority, and enter the following command in the Command window:

```
read path\scripts\jcatalog.sql
```

where *path* is your Adaptive Server Anywhere installation directory.

- ◆ Alternatively, enter the following command at a system prompt:

```
dbisql -c "uid=user;pwd=pwd"  
path\scripts\jcatalog.sql
```

where *user* and *pwd* identify a user with DBA authority, and *path* is your Adaptive Server Anywhere installation directory.

Loading the driver

Before you can use `jdbcConnect` in your application, you must load the driver. The simplest way to do this is with the following statement:

```
Class.forName("com.sybase.jdbc.SybDriver").newInstance()
;
```

Using the `newInstance` method works around issues in some browsers.

Supplying a URL for the server

In order to connect to a database via `jdbcConnect`, you need to supply a Universal Resource Locator (URL) for the database. An example was given in the section "Connecting from a JDBC client application" on page 507.

In the example, the connection was established as follows:

```
StringBuffer temp = new StringBuffer();
// Use the Sybase jdbc driver...
temp.append("jdbc:sybase:Tds:");
// to connect to the supplied machine name...
temp.append(_coninfo);
// on the default port number for ASA...
temp.append(":2638");
// and connect.
System.out.println(temp.toString());
conn = DriverManager.getConnection(temp.toString() ,
    _props );
```

The URL is put together in the following way:

```
jdbc:sybase:Tds:machine-name:port-number
```

The individual components are as follows:

- ◆ **jdbc:sybase:Tds** The Sybase `jdbcConnect` JDBC driver, using the TDS application protocol.
- ◆ **machine-name** The IP address or name of the machine on which the server is running. If you are establishing a same-machine connection, you can use **localhost**, which means the current machine
- ◆ **port number** The port number on which the database server is listening. The port number assigned to Adaptive Server Anywhere is 2638, and you should use that number unless there are specific reasons not to do so.

The connection string must be less than 253 characters in length.

Specifying a database on a server

Each Adaptive Server Anywhere server may have one or more databases loaded at a time. The URL as described above specifies a server, but does not specify a database. The connection attempt is made to the default database on the server.

You can specify a particular database by providing an extended form of the URL in one of the following ways.

Using the ServiceName parameter

```
jdbc:sybase:Tds:machine-name:port-number?ServiceName=DBN
```


The question mark followed by a series of assignments is a standard way of providing arguments to a URL. The case of **servicename** is not significant, and there must be no spaces around the = sign. The *DBN* parameter is the database name.

Using the RemotePWD parameter

A more general method is to provide additional connection parameters such as the database name, or a database file, using the **RemotePWD** field, as follows:

```
jdbc:sybase:Tds:machine-name:port-number?RemotePWD="dbf=f:\sybase\asademo.db"
```

By using the database file parameter **DBF**, you can start a database on a server using jConnect. By default, the database is started with `autostop=YES`. If you specify a DBF or DBN of **utility_db**, then the utility database will automatically be started.

 For information on the utility database, see "Using the utility database" on page 620.

Creating distributed applications

In a **distributed application**, parts of the application logic run on one machine, and parts run on another machine. With Adaptive Server Anywhere, you can create distributed Java applications, where part of the logic runs in the database server, and part on the client machine.

Adaptive Server Anywhere is capable of exchanging Java objects with an external, Java client.

The key task in a distributed application is for the client application to retrieve a Java object from a database. This section describes how to accomplish that task.

Related tasks

In other parts of this chapter, we have described how to retrieve several tasks that are related to retrieving objects, but which should not be confused with retrieving the object itself.

- ◆ "Querying Java objects" on page 486 describes how to retrieve an object into a SQL variable. This does not solve the problem of getting the object into your Java application.
- ◆ "Querying Java objects" on page 486 also describes how to retrieve the public fields and the return value of Java methods. Again, this is distinct from retrieving an object into a Java application.
- ◆ "Inserting and retrieving objects" on page 521 describes how to retrieve objects into server-side Java classes. Again, this is not the same as retrieving them into a client application.

Requirements for distributed applications

There are several tasks in building a distributed application.

❖ To build a distributed application:

- 1 Any class running in the server must implement the `Serializable` interface. This is very simple.
- 2 The client-side application must import the class, so that the object can be reconstructed on the client side.
- 3 Use the `sybase.sql.ASAUtils.toByteArray` method on the server side to serialize the object.
- 4 Use the `sybase.sql.ASAUtils.fromByteArray` method on the client side to reconstruct the object.

These tasks are described in the following sections.

Implementing the Serializable interface

Objects are passed from the server to a client application in what is called **serialized** form. For an object to be sent to a client application, it must implement the Serializable interface. Fortunately, this is a very simple task.

❖ To implement the Serializable interface:

- ◆ Add the words `implements java.io.Serializable` to your class definition.

For example, the Product class in the `jxmp\asademo` subdirectory implements the Serializable interface by virtue of the following declaration:

```
public class Product implements java.io.Serializable
```

Implementing the Serializable interface amounts to simply declaring that your class can be serialized.

The Serializable interface contains no methods and no variables. Serializing an object converts it into a byte stream which allows it to be saved to disk or sent to another Java application where it can be reconstituted, or **deserialized**.

A Java object that has been serialized in a database server, sent to a client application and deserialized, is identical in every way to its original state. Some variables in an object, however, either don't need to or, for security reasons, should not be serialized. Those variables are declared using the keyword `transient`, as in the following variable declaration.

```
transient String password;
```

When an object with this variable is deserialized, the variable will always contain its default value, null.

Custom serialization can be accomplished by adding `writeObject()` and `readObject()` methods to your class.

☞ For more information about serialization, see Sun Microsystems' *Java Development Kit (JDK)*.

Importing the class on the client side

On the client side, any class that retrieves an object has to have access to the proper class definition in order to use the object. To use the `Product` class, which is part of the `asademo` package, you must include the following line in your application:

```
import asademo.*
```

The *asademo.jar* file must be included in your CLASSPATH for this package to be located.

A sample distributed application

The *JDBCExamples.java* class contains three methods that illustrate distributed Java computing. These are all called from the `main` method. This method is called in the connection example described in "Connecting from a JDBC client application" on page 507, and is an example of a distributed application.

Here is the `getObjectColumn` method from the *JDBCExamples* class.

```
private static void getObjectColumn() throws Exception {
// Return a result set from a column containing
// Java objects
asademo.ContactInfo ci;
String name;
String sComment ;

if ( conn != null ) {
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(
"SELECT JContactInfo FROM jdbs.contact"
);

while ( rs.next() ) {
ci = ( asademo.ContactInfo )rs.getObject(1);
System.out.println( "\n\tStreet: " + ci.street +
"City: " + ci.city +
"\n\tState: " + ci.state +
"Phone: " + ci.phone +
"\n" );
}
}
}
```

The `getObject` method is used in the same way as in the internal Java case.

Older method

getObject and setObject recommended

The `getObject` and `setObject` methods remove the need for explicit serialization and deserialization that was needed in earlier versions of the software. The current section describes that older method, for users who are maintaining code that uses these techniques.

In this section we describe how one of these examples works. You can study the code for the other examples.

Serializing and deserializing query results

Here is the `serializeColumn` method of an old version of the `JDBCExamples` class.

```
private static void serializeColumn() throws Exception {
    Statement stmt;
    ResultSet rs;
    byte arrayb[];
    asademo.ContactInfo ci;
    String name;

    if ( conn != null ) {
        stmt = conn.createStatement();
        rs = stmt.executeQuery( "SELECT
            sybase.sql.ASAUtils.toByteArray( JName.getName() )
AS Name,
            sybase.sql.ASAUtils.toByteArray(
jdba.contact.JContactInfo )
            FROM jdba.contact" );

        while ( rs.next() ) {
            arrayb = rs.getBytes("Name");
            name = ( String
)sybase.sql.ASAUtils.fromByteArray( arrayb );
            arrayb = rs.getBytes(2);
            ci =
(asademo.ContactInfo)sybase.sql.ASAUtils.fromByteArray(
arrayb );
            System.out.println( "Name: " + name +
                "\n\tStreet: " + ci.street +
                "\n\tCity: " + ci.city +
                "\n\tState: " + ci.state +
                "\n\tPhone: " + ci.phone +
                "\n" );
        }
        System.out.println( "\n\n" );
    }
}
```

Here is how the method works:

- 1 A connection already exists when the method is called. The connection object is checked, and as long as it exists, the code is executed.
- 2 A SQL query is constructed and executed. The query is as follows:

```
SELECT
  sybase.sql.ASAUtils.toByteArray( JName.getName() )
  AS Name,
  sybase.sql.ASAUtils.toByteArray(
    jdba.contact.JContactInfo )
  FROM jdba.contact
```

This statement queries the *jdba.contact* table. It gets information from the **JName** and the **JContactInfo** columns. Instead of just retrieving the column itself, or a method of the column, the **sybase.sql.ASAUtils.toByteArray** function is used. This function converts the values to a byte stream so it can be serialized.

- 3 The client loops over the rows of the result set. For each row, the value of each column is deserialized into an object.
- 4 The output (**System.out.println**) shows that the fields and methods of the object can be used as they could in their original state.

Other features of distributed applications

There are two other methods in *JDBCExamples.java* that use distributed computing:

- ◆ **serializeVariable** This method creates a native Java object referenced by a SQL variable on the database server and passes it back to the client application.
- ◆ **serializeColumnCastClass** This method is like the **serializeColumn** method, but demonstrates how to reconstruct subclasses. The column that is queried (**JProd** from the **product** table) is of data type **asademo.Product**. Some of the rows are **asademo.Hat**, which is a subclass of the **Product** class. The proper class is reconstructed on the client side.

