

***Java Database  
Connectivity***  
***By Duncan Strand***  
*hc97ds1@dmu.ac.uk*

## **Table of Contents**

<b>TABLE OF CONTENTS</b>	<b>2</b>
<b>INTRODUCTION</b>	<b>3</b>
<b>WHAT IS JDBC</b>	<b>4</b>
<b>WHY WOULD I WANT IT?</b>	<b>4</b>
<b>WHAT CAN IT DO FOR ME?</b>	<b>4</b>
<b>DRIVERS</b>	<b>4</b>
<b>HOW</b>	<b>5</b>
<b>SQL PROCEDURES</b>	<b>6</b>
<b>TRANSACTIONS</b>	<b>7</b>
<b>SECURITY</b>	<b>7</b>
<b>APPLICATIONS</b>	<b>8</b>
<b>APPLETS</b>	<b>8</b>
<b>SERVLETS</b>	<b>8</b>
<b>EXAMPLES</b>	<b>8</b>
<b>REFERENCES</b>	<b>9</b>
<b>DBTEST</b>	<b>11</b>

## Introduction

I will describe what JDBC is and how it can be utilised within a Java program. A Java program includes Applets, applications and Servlets. I will briefly discuss the potential security implications of using JDBC. I will also include incomplete example code, and some fully functional example programs (which will be available on the web site). All code will be rendered in a different fixed width typeface like `this`.

I am assuming that the reader has some experience with Java and SQL, but not necessarily JDBC. I have included example code in Appendix A and will discuss the code within the body of the report. All code is available on the web site.

The web site address for this report is: [www.zeps.org.uk/ass/Year3Sem6/SOFT3010](http://www.zeps.org.uk/ass/Year3Sem6/SOFT3010)

## What is JDBC

JDBC (Java DataBase Connectivity) allows a Java program to store and retrieve data from any supported data source. All types of Java program can use JDBC, only applets have some security restrictions (I will discuss applets later).

A data source can be anything from a text file to a relational database. The location of the data source can be either a file on the users computer (applications only) or another computer connected via a network (applications, servlets and – with restrictions – applets).

Essentially JDBC is an API (application program interface) which allows programmers to access any data held by a DBMS (database management system), without knowing anything about the DBMS, and the method with which it stores data.

## Why would I want it?

Perhaps the most common use for JDBC is within e-commerce, where it can provide interaction between the visitor to a web site and the companies database. There are many websites where examples of this can be seen. Determining if the site is using Java for database retrieval purposes is difficult, as there are many different alternatives to Java and JDBC. The alternatives include PHP3 ([www.php.net](http://www.php.net)), and Perl ([www.perl.com](http://www.perl.com)), the Java equivalent of these is a servlet. I will discuss Servlets later.

Aside from the advantages for allowing the storage and retrieval of data via the Internet, JDBC has many positive applications for programs that are not intended for use online. Large numbers of application programs are involved in the storage, organisation and retrieval of data. I can say, from personal experience, that writing your own code to store data can be very time consuming, if however the programmer uses a database management system to store the data the programmer can save themselves a lot of time. In addition to simplifying the storage of the data the database management system can allow the programmer to query the data in a more complex manner than would have otherwise been reasonably feasible. However by implementing the storage via a database management system the programmers were tying themselves to a particular database system. To help solve this problem the ODBC (Open Database Connectivity) API was developed, this allowed the programmer to develop an application without worrying how the storage of the applications data is actually achieved.

With the introduction of Java the programmer finds themselves with a programming language with which they can create programs that can run on any supported computer platform (i.e. any operating system that has a virtual machine available). However because ODBC is not a standard part of all operating systems, nor, necessarily the same implementation and behaviour on all systems, Java is not capable of (directly) supporting ODBC. Fortunately Java sports an equivalent API called JDBC, and this is platform independent.

## What can it do for me?

There are many different practical applications for database access, some of the most obvious applications are web based. If a business needs to establish an online store they will need a database of the stock, and for storing the details of the customers orders. This can be achieved by using servlets that run on the server and dynamically create a web page when the user visits it – thus ensuring that the page contains the most up to date data.

## Drivers

To be able to access data from a data source there needs to be some mechanism that can interrogate the physical data file. The way this is achieved is by creating a software driver that knows how to communicate with one or more specific types of database. Surely though having different drivers could cause all sorts of problems with each driver working in a different manner? That could happen; however for a JDBC driver to be officially supported it must conform to the set standards that define how the driver should behave.

There are essentially two different types of JDBC driver, those that are pure Java and drivers that use native code.

An example of a native code driver is the JDBC-ODBC Bridge driver. This allows Java programs to use ODBC drivers. However by using the bridge the program loses its ability to run on any computer, because of this, and the additional set-up requirements the bridge is best suited to corporate Intranets.

Some of the reasons that the bridge was developed included the desire to remain compatible with old and unsupported DBMS and to help kick-start the development and use of JDBC.

## How

Before a program can start retrieving data from a database the program has to create a connection to the database. Regardless of the databases physical location a URL is used to indicate the location of the database. The JDBC guide states that recommended format of the URL is "jdbc:<subprotocol>:<subname>". So what is the significance of the component parts of the URL.

jdbc	This section is constant and doesn't change. It identifies that the URL contains information pertaining to a JDBC database connection.
subprotocol	This defines the type of database connection mechanism, and must be supported by one or more drivers. The specified protocol dictates the format of the subname.
subname	As stated the format of the subname varies according to the subprotocol. For example if the subprotocol is 'ODBC' then the subname is the name of the data source configured within the ODBC settings. The subprotocol can also specify a location on a network (as could the ODBC data source).

For security reasons many databases are protected by a password. Any user name and password should be specified when creating the connection. The example below creates a connection and logs in using a rather obvious username and password:

```
con = DriverManager.getConnection( url, "username", "password" );
```

Once a connection to the database has been established we can build a statement to issue to the database. There are several ways in which this can be done; both however have the same result. The first method is show below:

```
Statement stmt = con.createStatement(); // con is the DB connection
String query = "SELECT * FROM Addresses";
ResultSet rs = stmt.executeQuery(query);
```

With this example a statement is prepared with the createStatement() method. The SQL statement is then created and used as the parameter to the executeQuery() method, this returns a result set object. The programmer also has the option of specifying the SQL query at the time that the statement is created.

Once the executeQuery method has been executed we can start processing the data from the database, assuming any data has been returned.

With the above example the resulting data is stored in the class ResultSet. A simple method for retrieving and displaying data from an example database is:

```
while (rs.next()) {
    String s = rs.getString( 1 );
    String s1 = rs.getString( 2 );
    System.out.println( s + "\t" + s1 );
}
```

This loops until there are no more entries in the recordset. The current position in the recordset, or the place the cursor is currently pointing to, is moved to the next row when the next method is called. The next method returns true if there are further rows in the record set, false if there aren't any more.

When we want the data from the database to be manipulated within the program we must extract it from the database. Because there is no way that JDBC can determine the format of the data within the database it is up to the programmer to ensure that they use the correct method and datatypes when retrieving the data. If the datatype is unknown, or specific to the DBMS then the getObject method can be used. With the above example the data in the first column is in the database defined as an integer but fortunately Java can cope with this and convert it into a string. There are limitations however; for example it is not possible to convert an integer value from the database to a time value, when this happens the getXXX() function will throw a SQLException.

In addition to retrieving data the programmer can also store data in the database. Given that the programmer uses SQL to obtain data from the database the programmer must also use SQL to store it. The SQL statement to perform this is "INSERT INTO". When modifying the database the executeUpdate method is used. The example below inserts a row into the database, the values in the row are "Bob", and "Simmons".

```
String query = "INSERT INTO Addresses ( FirstName, LastName)
VALUES ( \"Bob\", \"Simmons\")";
Statement stmt = con.createStatement();
stmt.executeUpdate( query );
```

As well as adding data the programmer may also need to modify data in the database without taking the steps of deleting the row the data is in, and the re-inserting it, the SQL "UPDATE" statement can be used to do this.

As mentioned above the Statement interface is used to construct and execute an SQL query. In addition there is a subinterface PreparedStatement, and this also has a subinterface called CallableStatement.

The prepared statement interface allows greater efficiency when the same SQL statement needs to be used multiple times, this is particularly useful when adding or modifying data in the database. The execution of the query will also be faster because the SQL statement is precompiled, this means that the DBMS does not need to recompile the query should it be reused. When using the prepared statement interface the SQL string can be constructed with missing values that can be changed later, these missing values are represented with a '?'. For example "SELECT \* FROM Addresses WHERE Name = ?". To set this query so that it matches all people with the name 'Janet' the setString method is used.

```
PreparedStatement stmt = con.prepareStatement("SELECT *
FROM Addresses WHERE FirstName = ?");
stmt.setString( 1, "Janet" );
ResultSet rs = stmt.executeQuery();
```

Using the prepared statement does not require a radical change in the code previously used, and there is still the same number of lines in the code. However the programmer can modify the query and run it again by using the setString method, and re-running the query.

To change the '?' placeholders there are a series of set methods, for example, setInt or setString, these methods take the index of the placeholder to change, and the value to change it to. Once a value has been set it remains the same until it is changed again or cleared using the clearParameters method.

## SQL Procedures

As previously mentioned the PreparedStatement interface has a subinterface called CallableStatement, this interface allows the programmer to call SQL procedures. An SQL procedure is a group or set of SQL statements that perform a specific set task. Almost all DBMS systems support SQL procedures but there is some variation in the capabilities and the syntax between different systems (HAMILTON, G 1998). This can potentially cause problems as the Java program may not be aware of the abilities of the DBMS capabilities and the program may try to create an unsupported procedure. This is not a problem if the program is only intended to work with a specific DBMS.

To create and store an SQL procedure we can use the Statement interface and pass a string that contains the necessary SQL for the procedure, for example:

```
String createProcedure = "create procedure SHOW_SUPPLIERS " +
"as " +
"select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME " +
"from SUPPLIERS, COFFEES " +
"where SUPPLIERS.SUP_ID = COFFEES.SUP_ID " +
"order by SUP_NAME";
Statement stmt = con.createStatement();
stmt.executeUpdate( createProcedure );
(HAMILTON, G 1998)
```

This creates a procedure that can be called at a later time and by any user who can connect to the database (for now - ignoring user privileges). To call the procedure the call is prepared

using the CallableStatement interface and the procedure executed using either the executeQuery or the executeUpdate methods, this allows the procedure to either return a result set or an update count. To call the procedure shown above the following code can be used:

```
CallableStatement cs = con.prepareCall( "{call SHOW_SUPPLIERS}" );  
ResultSet rs = cs.executeQuery();
```

Because CallableStatement is a sub interface of PreparedStatement, CallableStatement can use input parameters. The following example allows the programmer to call the same procedure, but changing the procedure parameters as needed (assuming that the procedure itself has been changed to accept parameters).

```
CallableStatement cs = con.prepareCall( "{call SHOW_SUPPLIERS( ? , ? )}" );
```

In addition to passing parameters into a procedure we can also retrieve values that are passed out of the procedure. Before calling such a procedure however the types of the data passed out must be registered. To register an out parameter the registerOutParameter method is called, this takes the position of the parameter and the type. Optionally it can also take the scale of the number, the number of digits to the right of the decimal point and a typeName which is the name of a SQL structured type. Once the query has been executed the parameters passed out can be obtained by using the relevant get methods that are defined in the CallableStatement interface.

SQL procedures can potentially return more than one result set, if this is likely to happen the execute method should be used. By using this method in conjunction with getResultSet or getUpdateCount to retrieve the result, and getMoreResults to move to the next result.

## Transactions

By standard when the executeUpdate method is called the changes are immediately made, there are many scenarios where this is undesirable.

To disable the automatic updating of the database the setAutoCommit method, when this is set to true the database is not updated until the commit method is called.

With any database available over a network it is quite possible that more than one person can be working with the database at any time. JDBC does not need to deal with this as it handled by the JDBC driver, as any record locking mechanism is database specific, however by using transactions the period with which any part of the database is locked can be extended.

## Security

Security issues with JDBC vary according to the nature of the program; for example an applet has different security considerations than say an application.

The JDBC Guide states that an applet cannot access any arbitrary host and that an untrusted applet cannot access files on the local computer. A Java applet can only open a database connection with the server that the applet was initially served from. This is a fairly restrictive for any serious development purposes, but it is within the general specification of the applet security model. If an applet is trusted, however, the security restrictions are lifted, and the applet can access the desired server. In a typical web application this is not ideal because the user has to specifically show that they trust the applet, in many cases the user would become suspicious of the applet and most likely decide not to trust it. Within an Intranet scenario the issue of trust is different because the source of the applet is from within the company.

For all other java applications (i.e. applications and servlets) there are no security restrictions, the program has free access to the files on the local computer (therefore including local database connections) and can connect to any server the program chooses.

With database access effectively allowing users to modify data on one of your servers it is important to ensure that the system is secure. If the database contains private or personal data the access to the database should be controlled with a user name and password. If this is the case then an applet should not be used as the user name and password would either need to be entered by the user or compiled into the Java class file.

With a servlet that is running on the webserver the servlet code can contain username and password because the visitor to the website cannot view the code. There are occasions where this is not possible or desirable as, for example, with a database of users the username and password list maybe changing frequently and it would not be possible to re-compile the code.

## Applications

Traditionally applications have been written in programming languages like C++. Previously there were no commonly used languages that provided support for database access (Delphi and Visual C++ are extensions upon the original language and are platform specific). With Java however this has changed database access is included as a standard part of the language. An example JDBC application can be found in appendix A.

## Applets

As I have already mentioned applets have security restrictions that severely restrict the database access that an applet can have. Essentially it can only open a database connection with the server that sent the applet to the users computer. The only manner in which an applet can circumvent these restrictions is if the user dictates that they trust the applet.

An applet can request different security privileges from the user, and unlike previous versions the Java 1.2 security model can allow only certain types of privileged access. However if a website were built with an applet that needs to request the required permissions the visitors to the website would (quite rightly) be concerned at the possible security implications of allowing the applet to query a remote database. I would recommend that all other possible alternatives be explored before resorting to using an applet with JDBC capabilities.

## Servlets

The glossary on the Java2 Enterprise Edition home page defines a servlet as "*A Java program that extends the functionality of a Web server, generating dynamic content and interacting with Web clients using a request-response paradigm.*".

An experienced web developer could well argue that this offers nothing new over previously available solutions like php and Perl. Although both php and Perl are platform independent there are certain restrictions on their portability, whereas with Java there are no such restrictions that effect the developer. In addition Java servlets operate and support the same functionality regardless of the webserver used.

Servlets offer the same capabilities as both php and Perl. The security restrictions placed on a servlet are the same as with a Java application, i.e. it is free to access the computers file system and establish database connections with other computer via a network.

The following code is an example servlet:

```
Public class HelloServlet extends HttpServlet {
    Public void doGet( HttpServletRequest request,
HttpServletRequestResponse, response) throws ServletException, IOException
    {
        response.setContentType( "text/plain" );
        PrintWriter out = response.getWriter();
        out.println( "Hello World!" );
    }
}
```

Whilst this clearly does not involve any database activity (it display the 'Hello World' message) it does show one important difference between a servlet and application. With a servlet the entry point is the doGet method (and also the doPost method) is the entry point to the program.

## Examples

Example code, and the database it works with can be viewed at this website: [www.zeps.org.uk/ass/Year3Sem6/SOFT3010/JDBC](http://www.zeps.org.uk/ass/Year3Sem6/SOFT3010/JDBC)

## References

HAMILTON, G (1998) JDBC Database Access with Java Addison-Wesley

<b>Title</b>	<b>URL</b>	<b>Verified</b>
JDBC Homepage	<a href="http://java.sun.com/products/jdbc/">java.sun.com/products/jdbc/</a>	8/5/2000
JDBC Guide	<a href="http://java.sun.com/products/jdk/1.3/docs/guide/jdbc/spec/jdbc-spec.frame.html">java.sun.com/products/jdk/1.3/docs/guide/jdbc/spec/jdbc-spec.frame.html</a>	8/5/2000
Java Enterprise Edition	<a href="http://java.sun.com/j2ee/">java.sun.com/j2ee/</a>	8/5/2000
Java Servlets White Paper	<a href="http://java.sun.com/products/servlet/whitepaper.html">java.sun.com/products/servlet/whitepaper.html</a>	8/5/2000

# ***Appendix A***

## ***Code Samples***

## DBTest

```
import java.sql.*;

public class DBTest
{
    public static void main(String args[]) {
        String url = "jdbc:odbc:JDBCTest";
        Connection con;

        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con = DriverManager.getConnection( url );

            Statement stmt = con.createStatement();
            String query = "SELECT * FROM Addresses";
            ResultSet rs = stmt.executeQuery(query);

            System.out.println( "ID\tName" );

            while (rs.next()) {
                String s = rs.getString( 1 );
                String s1 = rs.getString( 2 );
                System.out.println( s + "\t" + s1 );
            }

        }
        catch(SQLException ex) {
            System.err.println("SQLException: " +
ex.getMessage());
        }
        catch(ClassNotFoundException ex) {
            System.err.println("ClassNotFoundException: "
+ ex.getMessage());
        }
    }
}
```