Workshop III: Java Database Connectivity (JDBC)

Author: Gergely Mátéfi

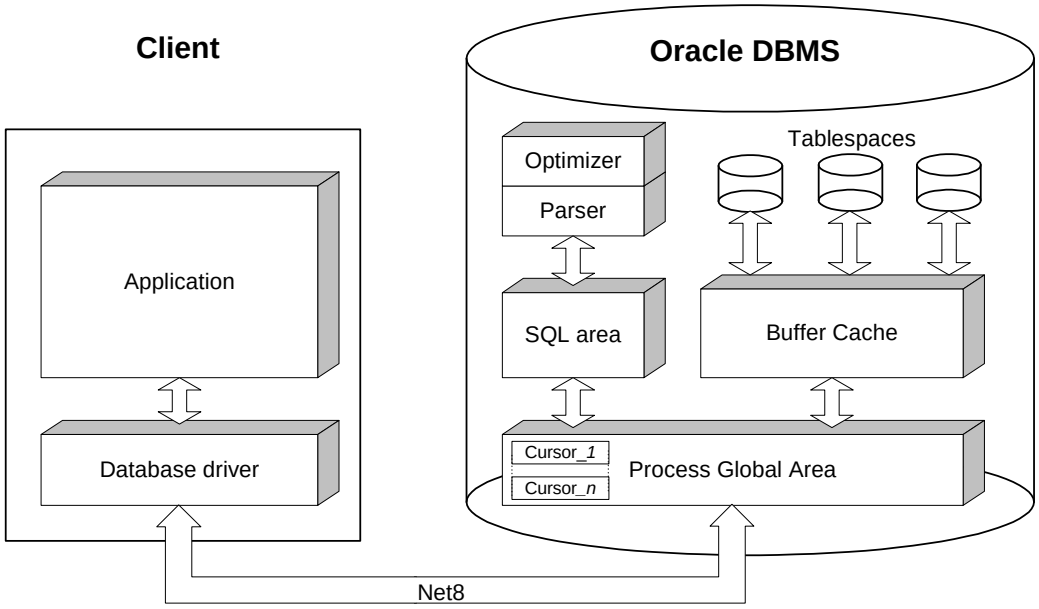
Introduction

Java Database Connectivity (JDBS) is a manufacturer-independent *de facto* standard of Java-based database access. The JDBC application programming interface (API) contains Java classes and interfaces which provide low-level access to relational databases, such as: connecting, executing SQL statements, processing results. Interfaces are implemented by the own drivers of the manufacturers. According to the Java principles – the drivers must only be available at runtime, thus application developers are able to create the Java application independently from the database management (DBMS) system.

The aim of the present workshop is to introduce database-related, client-server application development in a Java and JDBC environment. In the first chapter the client-server architecture will be described; which is followed by the enumeration of the main JDBC language elements. Finally, the usage of JDBC will be illustrated through a real-life example. According to the workshop environment the workshop is limited to introducing the JDBC version 1.2 API only.

Database management in a client-server architecture[1]

In client-server architecture the application running on the client connects to the DBMS through a network, using the driver provided by the manufacturer. Depending on the application the driver might be a C library, ODBC or JDBC.



By authenticating himself towards the DBMS the client has to create a database-connection (session) before starting to execute database operations. During the session creation in the Oracle system the DBMS allocates resources for the session, by reserving memory (Process Global Area – PGA) and by starting a server process[2].

During the lifecycle of the session the client can initiate database operations which are forwarded to the DBMS in the form of SQL statements. The SQL statement is processed by the DBMS in several steps. At the processing start the server process separates a memory area within the PGA to store processing information – such as the compiled SQL statement, the actual position within the result set (see below). The descriptor of the separated memory area is called cursor; the start of processing is called opening a cursor. A session might have more than one opened cursor at a time.

The first step of processing is the parsing of the SQL statement, where the DBMS compiles the string containing the statement and checks the valid access privileges to the affected database objects. Then an execution plan is created by the Optimizer for the successfully compiled statement. The execution plan contains the steps of physically selecting the rows affected by the statement: it specifies the starting table, the indexed to be used, how the selection is done. Since parsing and creating execution plans requires a lot of resources, the execution plans of the latest SQL statements are stored in the DBMS cache (SQL area).

---

[1] The basic concepts will be introduced by using the (simplified) operation of Oracle RDBMS, however these concepts are not Oracle-specific

[2] Server processes can be shared, as well

Usually a database-related application employs a few SQL statements with a specified structure but different parameters. A billing software for example queries the same data from different customers; therefore it is only the customer ID that changes from statement to statement. Therefore SQL enables calling these statements by using parameters:

```
SELECT NAME, ADDRESS, TAX_NR FROM CUSTOMER WHERE CUSTOMER_ID = ?
```

An SQL statement with parameters is compiled by the DMBS during processing it for the first time, in latter cases there is no need to re-compile the statement. Cache usage is enabled by the *parameters being substituted only after parsing and execution plan creation*.

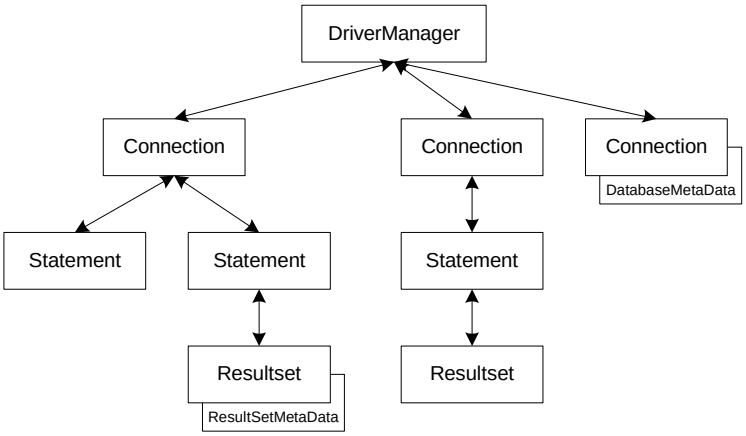
After the execution plan creation and the parameter substitution the SQL statement is *executed*. In SELECT type queries the selected lines logically create a *result table*, whose lines can be retrieved by the client one-by-one[3], using the *fetch* operation. After retrieving the result table, or finishing the transaction (commit / rollback) the memory area allocated for the processing is freed, and the *cursor is closed*.

The JDBC 1.2 API

The basic framework for data access

The JDBC API is a set of Java classes and interfaces. The most important classes and interfaces are:

- java.sql.DriverManager is a class responsible for URL resolution and creating new database sessions;
- java.sql.Connection is an interface representing a database connection;
- java.sql.DatabaseMetaData provides (meta)information on the database;
- java.sql.Statement controls SQL statement execution;
- java.sql.ResultSet is an interface providing access to a given query results
- java.sql.ResultSetMetaData is an interface providing meta-information on the result table



[3] In order to be more efficient the database driver might fetch more lines in a batch-processing fashion

Managing database connections

The management of JDBC drivers and the creation – closing of connections is done by the **`java.sql.DriverManager`** class. The attributes and methods of DriverManager are static, thus there is no need to instantiate the class in the application. A new connection is created in the DriverManager using the following statement:

```
Connection con = DriverManager.getConnection(url,
                                "myLogin", "myPassword");
```

The first parameter of the `getConnection` method is the URL string that identifies the database, the second and the third parameters are the name and the password of the database user. The content of the URL is database-dependent, according to the convention its structure is as follows:

```
jdbc:<subprotocol>:<subname>
```

where `<subprotocol>` identifies the mechanism used to connect to the database, and `<subname>` contains the parameters related to the mechanism specified in the `<subprotocol>` part. For example connecting to the ODBC data source identified by „Fred” can be done by the following statements:

```
String url = "jdbc:odbc:Fred";
Connection con = DriverManager.getConnection(url, "Fernanda", "J8");
```

If the URL requested by the driver contains username and password already, the second and the third parameters of the method might be omitted. By calling the method `getConnection` DriverManager starts querying every single *registered* JDBC drivers and creates the database connection using the first driver which is able to resolve the URL. After carrying out the required operations the connection can be closed using the **`Connection.close`** method of the DriverManager. Upon the deletion of the `Connection` object (garbage collection) the method `close` is called automatically.

Before use, all drivers must be *loaded* and *registered* at the DriverManager. Usually, application developers only have to take care about loading the driver; the drivers automatically register themselves in their static initialization method. The easiest way to load a driver is to use the **`Class.forName`** method[4], for example:

```
Class.forName("oracle.jdbc.driver.OracleDriver")
```

Due to security reasons and *applet* might only use drivers that are located on the local computer, or have been downloaded from the same address as the applet code. Unlike applets from „un-trusted sources”, „trusted” applets are run as complete programs by JVM, without any constraints.

Executing SQL statements

Simple SQL statements are executed using **`Statement`** the interface. First the class implementing the interface must be *instantiated*, then the relevant (SQL statement-dependent) *execution method* of the instance must be called in order to execute an

[4] In JDK 1.1.x due to a design error the static methods of a class loaded by the Class.forName method sometimes cannot be run; in these cases the JDBC driver must be registered using the registerDriver method of the DriverManager. The error usually occurs under Internet Explorer 4.x but not under Netscape Navigator 4.x

SQL statement. The `Statement` instance can be created by calling the **`createStatement`** method of the `Connection` instance representing the database connection:

```
Statement stmt = con.createStatement();
```

The *execute* series are the most often used of `Statement`'s methods:

- **`executeQuery`**: executes the SQL query passed on as parameter, and returns a result table (`ResultSet`). The method is used to execute query (SELECT) statements.
- **`executeUpdate`**: executes the SQL statement passed on as parameter, and returns the number of rows affected by the modification. This method can be used to execute both data manipulation (DML) and data definition (DDL) statements. In case of DDL statements the return value is 0.
- **`execute`**: executes the SQL statement passed on as parameter. This method is the generalization of the previous two. The return value is True; in case the return value is of type `ResultSet` it can be retrieved by the **`Statement.getResultSet`** method. The number of rows affected by the modification can be retrieved by the **`Statement.getUpdateCount`** method.

In the following example we are creating a table containing all the bill data for the billing software of CoffeeBreak Company:

```
int n = stmt.executeUpdate("CREATE TABLE COFFEES ( " +
      "COF_NAME VARCHAR(32), " +
      "SUP_ID   NUMBER(8), "   +
      "PRICE    NUMBER(6,2), " +
      "SALES    NUMBER(4), "   +
      "TOTAL    NUMBER(6,2)");
```

Once the business is started, the purchases can be queried using the following statement:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM COFFEES");
```

The execution of a statement is finished when all result tables are processed (all lines of the result tables are retrieved). However, statement execution can be terminated manually by using the **`Statement.close`** method. By repeatedly calling the execution method of a `Statement` object the earlier, unfinished execution of the same object is automatically terminated.

Handling *SQL statements with parameters* somewhat differs from handling simple SQL statements. In JDBC SQL statements with parameters are represented by the **`PreparedStatement`** interface. The interface is created in a way similar to `Statement`, by calling the **`prepareStatement`** method of the `Connection` instance representing the connection and supplying the *SQL statement with parameters*:

```
PreparedStatement updateSales = con.prepareStatement(
            "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

The parameter values – indicated with question marks – of the statement stored in the `PreparedStatement` can be set by using the **`setXXX`** method family. For the execution the following methods (introduced earlier at the `Statement` class) might be used without any arguments specified: `executeQuery`, `executeUpdate` and `execute`. The following example illustrates data entry into the `COFFEES` table using SQL statements with parameters:

```
PreparedStatement updateSales;
String updateString =
 "update COFFEES set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
                     "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
      updateSales.setInt(1, salesForWeek[i]);
      updateSales.setString(2, coffees[i]);
      updateSales.executeUpdate();
}
```

The **`setXXX`** methods are expecting two arguments. The first argument is the index of the SQL parameter to be set; the parameters in the SQL statements are indexed from left to right, starting with 1. The second argument is the value to be set. Please note, that JDBC *does not perform implicit type conversion* for the input parameters; therefore it is the programmer's responsibility to provide right data types for the database-manager. Null values can be set by using the method **`PreparedStatement.setNull`**. A set parameter can be reused for multiple executions of the SQL statement.

Stored procedures and methods can be called by using the **`CallableStatement`** interface. The `CallableStatement` interface can be instantiated by calling the **`prepareCall`** method of the `Connection` instance representing the connection. The `CallableStatement` interface is derived from the `PreparedStatement` interface thus the input parameters (IN) can be set using the `setXXX` methods. Output parameters (OUT) have to be registered by specifying their type before execution by using the **`CallableStatement.registerOutParameter`** method. As it can be inferred from the following example, after the execution output parameters can be retrieved using the **`getXXX`** method family.

```
CallableStatement stmt = conn.prepareCall("call getTestData(?,?)");
stmt.registerOutParameter(1,java.sql.Types.TINYINT);
stmt.registerOutParameter(2,java.sql.Types.DECIMAL);
stmt.executeUpdate();
byte x = stmt.getByte(1);
BigDecimal n = stmt.getBigDecimal(2);
```

Similar to the `setXXX` methods `getXXX` methods *do not perform any type conversions*: it is the responsibility of the programmer to ensure that the data types provided by the database are in sync with the `registerOutParameter` and the `getXXX` methods.

Managing result tables

Query results can be accessed through the **`java.sql.ResultSet`** class which is instantiated by the `executeQuery` or `getResultSet` method of the `Statement` interfaces. It is only the *actual* row (marked by the cursor) of the result table represented by `ResultSet` that can be accessed. Initially, the cursor is pointing *before* the first row and the method **`ResultSet.next`** can be used to position the cursor on the next row[5]. Method `next` returns False in case the cursor passed the last row, otherwise it returns True.

[5] Positioning the cursor on the previous row – or on a particular row – is only possible in JDBC 2.0 (in case it is also supported by the driver)

Field values of the actual row can be retrieved using the method family `getXXX` [6]. `getXXX` methods can reference fields in two ways: by column indexes and column names. In SQL queries columns are indexed from left to right, starting with 1. Referencing by column names is less efficient due to runtime mappings but is a more comfortable solution. Unlike the `getXXX` methods of `PreparedStatement` and `CallableStatement`, the `ResultSet.getXXX` methods *perform automatic type conversions*. In case type conversion is not possible (for example by calling the method `getInt` for a field of type `VARCHAR` containing the string "`foo`") an `SQLException` exception is raised. In case a field *contains SQL NULL* value `getXXX` returns zero or Java null depending on the `getXXX` method. *After* retrieving the field value method **`ResultSet.wasNull`** can be used to check whether the retrieved value was derived from an SQL NULL or not[7]. Usually the programmer does not have to deal with closing the result table since that is automatically closed when the `Statement` is finished. However, the result table can be closed manually as well by the **`ResultSet.close`** method.

Meta-information on the result tables can be accessed through the **`ResultSetMetaData`** interface. The object implementing the interface is returned by the method **`ResultSet.getMetaData`**. The `ResultSetMetaData` method **`getColumnCount`** returns the number of columns, while **`getColumnName(int column)`** returns the name of the column having the index specified.

The following example illustrates the use of `ResultSet` and `ResultSetMetaData`. The first column is `integer`; the second is `String`, while the third is an array of bytes.

```
Statement stmt = conn.CreateStatement();
ResultSet r = stmt.executeQuery("SELECT a, b, c FROM table1”);
ResultSetMetaData rsmd = r.getMetaData();
for (int j = 1; j <= rsmd.getColumnCount(); j++) {
      System.out.print(rsmd.getColumnName(j));
}
System.out.println;
while (r.next()) {
      // Printing fields of the actual row
      int i = r.getInt("a");
      String s = r.getString("b");
      byte b[] = r.getBytes("c");
      System.out.println(i + " " + s + " " + b[0]);
}
stmt.close();
```

Error handling

In case any kind of error occurs during the database connection, on Java-level an **`SQLException`** is generated. Method **`SQLException.getMessage`** returns the error message text, **`SQLException.getErrorCode`** returns the error code, and **`SQLException.getSQLState`** returns the state description that complies with the X/Open SQLstate convention.

---

[6] For a list of `getXXX` methods please refer to the Appendices.

[7] Null check before retrieving a field value is not supported by all DBMS therefore it was excluded from JDBC 1.2 API.

Transaction handling

Database connections represented by the class Connection are in *auto-commit mode, as a default*. It means that every SQL statement (`Statement`) runs as a unique transaction and is committed after its execution (commit). This default setting can be overwritten using the method **`Connection.setAutoCommit(false)`**. In this case the transaction has to be committed or rolled back from the program using the method **`Connection.commit`** and **`Connection.rollback`**, as it is illustrated in the following example:

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
      "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
      "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Database information

Database-related information (metadata) can be accessed through the **`DatabaseMetaData`** interface. The class implementing the interface is returned by the **`getMetaData`** method of the `Connection` instance representing the connection.

```
DatabaseMetaData dbmd = conn.getMetaData();
```

Depending on the information requested the `DatabaseMetaData` interface methods return with either a simple Java type or a `ResultSet`. The table below lists a few important methods.

Method name	Return value	Description
`getDatabaseProductName`	`String`	Name of the database product
`getDatabaseProductVersion`	`String`	Database product version number
`getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`	`ResultSet`	Lists the tables matching the provided patterns

The Oracle JDBC drivers

The two client-side drivers implemented by Oracle are: the *JDBC OCI Driver* and the *JDBC Thin Driver*.

The **JDBC OCI Driver** implements the JDBC methods as OCI library calls. The Oracle Call Interface (OCI) consists of a set of standard C-language software APIs (database drivers) which provide a low-level interface to the DBMS services for high-level development tools. By calling a method in JDBC (e.g. statement execution) the JDBC OCI Driver is forwarding the call to the OCI layer, which is then transmitted to the database manager via SQL*Net or Net8 protocols. Due to the C library calls the JDBC OCI Driver is platform- and operating system dependent; however its native coding makes it more efficient than the purely Java-based Thin Driver.

The **JDBC Thin Driver** is entirely written in Java. The Thin Driver contains a simplified, TCP/IP-based implementation of SQL*Net/Net8 protocol, therefore herein a JDBC method call is immediately transmitted to the database manager. Due to the pure Java implementation the JDBC Thin Driver is platform-independent and can be downloaded together with the Java applet. However, since the implementation is simplified the JDBC Thin Driver does not provide all OCI features (e.g. encrypted communication, non-IP protocols etc. are not supported).

For addressing the databases – matching the JDBC convention – Oracle uses the following URL structure:

```
jdbc:oracle:drivertype:user/password@host:port:sid
```

where `drivertype` can be *oci7*, *oci8* or *thin*; `host` is the DNS name of the database-server; `port` is the port number of the server-side TNS listener; and `sid` is the database identifier. Username and password can also be provided in the second and third parameters of the `getConnection` method, and in this case the `user/password` part can be excluded from the URL structure.

A demo Applet

The applet included below provides access to a simple library system. The driver name and the database URL are imported as parameters from the embedding HTML page; database connection is created by clicking on the *Connect* button. By pressing the *Search* button book titles starting with the string specified in the *titleField* are listed in the *result* area. The constructor located at the beginning of the source creates a GUI and assigns event handlers to the buttons; *Print* methods are used to print different objects (String, SQLException, and ResultSet); while the *actionPerformed* method handles the on-button-click events and *destroy* terminates the possibly open connection when the applet is closed.

```
import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import java.applet.Applet;

public class Library extends Applet implements ActionListener{
  Button    connectButton = new Button("Connect");
  Button    clearButton   = new Button("Clear");
  TextField titleField    = new TextField(30);
  Button    listButton    = new Button("Search");
  TextArea  result        = new TextArea(20,100);
  Connection con = null;

  public Library() {
    super();
    Panel listPanel = new Panel();
    listPanel.add(new Label("Title: "));
    listPanel.add(titleField);
    listPanel.add(listButton);
    add("North",listPanel);
    Panel resultPanel=new Panel(new BorderLayout());
    resultPanel.add("North", new Label("Output:"));
    result.setEditable(false);
    result.setFont(new Font("Monospaced", Font.PLAIN, 10));
    resultPanel.add("South", result);
    add("Center", resultPanel);
    Panel controlPanel = new Panel();
```

```
  controlPanel.add(connectButton);
  controlPanel.add(clearButton);
  add("South",controlPanel);

  connectButton.addActionListener(this);
  clearButton.addActionListener(this);
  listButton.addActionListener(this);
  validate();
}

public static void main (String args[])  {
  Library myLibrary = new Library();
  Frame frame = new Frame(myLibrary.getClass().getName());
  frame.add("Center",myLibrary);
  frame.setSize(500,700);
  frame.show();
}

private void Print(String text) {
  result.append(text+"\n");
}

private void Print(SQLException e) {
  while (e!=null) {
    Print("SQLException occured:" +
       (e instanceof SQLWarning ? "WARNING" : "ERROR"));
    Print("SQLState: "+e.getSQLState());
    Print("Message: "+e.getMessage());
    Print("Code: " + e.getErrorCode());
    e=e instanceof SQLWarning ? ((SQLWarning)e).getNextWarning()
                              : e.getNextException();
  }
}

private void Print(ResultSet rset) throws SQLException {
  ResultSetMetaData rsmd = rset.getMetaData();
  String s = "";
  int i, colNum = rsmd.getColumnCount();
  for (i=1; i<=colNum; i++)
    s += rsmd.getColumnLabel(i) + "\t\t";
  Print(s);
  while (rset.next()) {
    s = "";
    for (i=1;i<=colNum; i++)
      s += rset.getString(i) + "\t\t";
    Print(s);
  }
}

public void actionPerformed(ActionEvent evt)
{
  if (evt.getSource()==connectButton) {
     if (con!=null) {
       Print("Already connected.");
       return;
     }
     try {
       String driverName = getParameter("driverName");
       try {
         Class.forName(driverName);
         Print(driverName + " loaded.");
```

```
          }
          catch (ClassNotFoundException e) {
            Print(driverName + " not found. ");
            return;
          }
          if (java.lang.System.getProperty("java.vendor").equals(
            "Netscape Communications Corporation")) {
              netscape.security.PrivilegeManager.enablePrivilege(
                "UniversalConnect");
          }
          else if (java.lang.System.getProperty("java.vendor").equals(
            "Microsoft Corp.")) {
               DriverManager.registerDriver(
                new oracle.jdbc.driver.OracleDriver());
          }

          String url =getParameter("url");
          Print("Connecting to " + url);
          con=DriverManager.getConnection(url);
          DatabaseMetaData dbmd = con.getMetaData();
          Print("DBMS name: " + dbmd.getDatabaseProductName() +
                " version "+ dbmd.getDatabaseProductVersion());
        }
        catch (SQLException e) { Print(e); }
      }

      else if (evt.getSource()==listButton) {
        if (con == null) {
           Print("Not connected.");
           return;
        }
        try {
          Statement stmt = con.createStatement();
          ResultSet rset = stmt.executeQuery(
             "SELECT isbn, author, title " +
             "FROM book " +
             "WHERE title LIKE '"+ titleField.getText() + "%'" +
             "ORDER BY title, author");
          Print(rset);
          stmt.close();
        }
        catch (SQLException e) { Print(e); }
      }

      else if (evt.getSource() == clearButton) {
        result.setText("");
      }
    }

    public void destroy() {
      try {
        con.close();
      }
      catch (SQLException e) {}
    }
}
```

The HTML file embedding the applet:

```
<html><body>
<applet code="Library.class" archive=classes111.zip width="500" height="300">
<param name=driverName value="oracle.jdbc.driver.OracleDriver">
```

```
<param name=url value="jdbc:oracle:thin:scott/tiger@rapid.eik.bme.hu:1521:szglab">
Sorry, your browser can't display Java applets.
</applet></body></html>
```

Please note that certain security measures must be taken into consideration when running an applet. By default, an applet without a certificate can only connect to the server from which it has been downloaded. During the applet development phase however, depending on the run-time environment, this security setting of the JVM can be turned off[8].

In the workshop environment under Netscape Communicator 4.77 browser the applet might ask the user to disable network connection limitations using the following method: `PrivilegeManager.enablePrivilege("UniversalConnect")`.

Getting ready

Apart from universal programming skills JDBC specific knowledge and SQL query experience is also required for successfully completing the workshop exercises. Due to the large number of possible error conditions the workshop requires thorough preparation. Preparation materials and configuration files are available under the following address: http://db.bme.hu/java/jdbc. It is suggested that you follow these steps!

0. Take a look at the Java recap, if necessary. Read through the JDBC help. (JDBC knowledge will be assessed during a short test before the workshop.)
1. Create the required environment on the web server. Check the JDK version number (`java -version`). Create a `~/public_html` library (if necessary) and copy the JDBC Thin Driver matching the operating system and the DBMS.
2. Copy the demo applet, the embedding HTML file and the `build.sh` script to facilitate the compilation. Compile the applet. Set the necessary read privileges for the web server.
3. Enable universal network connection for applets in your browser. Test the demo applet (possible exceptions are displayed in the Java Console of the browser, debugging can also be performed from here).
4. Modify the demo applet in a way that one can search for author names and ISBN, as well.

References

The JDBC API Version 1.20, Sun Microsystems Inc., 1997 (available at http://www.javasoft.com)

S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner: *JDBC 2.0 API Tutorial and Reference, Second Edition: Universal Data Access for the Java 2 Platform*, 1999 (available at http://www.javasoft.com)

S. Kahn: *Accessing Oracle from Java*, Oracle Co., 1997. (available at http://www.oracle.com)

Oracle8™ Server Concepts, Release 8, Oracle Co.

Nyékiné et al. (szerk*): Java 1.1 útikalauz programozóknak*, ELTE TTK Hallgatói Alapítvány, 1997.

[8] Under Internet Explorer 4.x it can be enabled under security settings, under Netscape Communicator 4.x add the following row `user_pref("signed.applets.codebase_principal_support", true);` to the `<Netscape dir>\Users\<Username>\prefs.js` file.

Appendix A: Accessing Oracle data types from JDBC

Java type	Access method	CHAR	VARCHAR2	NUMBER	DATE
byte	getByte	x	x	x	
short	getShort	x	x	x	
int	getInt	x	x	x	
long	getLong	x	x	x	
float	getFloat	x	x	x	
double	getDouble	x	x	x	
java.Math.BigDecimal	getBigDecimal	x	x	x	
boolean	getBoolean	x	x	x	
String	getString	**X**	**X**	x	x
java.sql.Date	getDate				x
java.sql.Time	getTime				x
java.sql.TimeStamp	getTimeStamp				**X**

Signage:
x: the `getXXX` method *can be used* to access that particular SQL type
X: the `getXXX` method *is recommended* to access that particular SQL type

Appendix B: Brief JDBC history

JDBC has been changing a lot during its history. Originally, at the beginning of 1997 *JDBC 1.0 API* was introduced as a simple, manufacturer-independent, unified – and therefore featuring minimal functionality – programming interface supplementing Java Development Kit (JDK) 1.0. In the latter JDK 1.1 JDBC was already thoroughly integrated, with its classes being part of the Java base classes (`java.sql.*`). The JDBC 2.0 which was presented with JDK 1.2 contains two packages. The `java.sql` package containing the *JDBC 2.0 Core API* enhances the original JDBC API with new functionality. New functionalities include: positionable result tables, result table modification support with direct JDBC methods, handling of SQL99 specification standard basic (BLOB, CLOB, Array) and user-defined (User Defined Type, UDT) data types. Apart from the DriverManager architecture aiming to directly handle the drivers the *JDBC 2.0 Optional Package*[9] (`javax.sql`) introduces the data source-based (DataSource) access model, which provides connection pooling and distributed transaction handling. As of now,[10] the JDBC 3.0 API is fine-tuning the earlier modifications and integrates JDBC more closely with other Java 2 technologies (such as Connector Architecture etc.).

[9] Under its former name: *JDBC Standard Extension API*
[10] In January, 2002.