BELL College
BSc Applied Computing

Software Development

4. Working with Databases using JDBC

Contents

INTRODUCTION 2
CREATING A DATABASE CONNECTION 3
JDBC DRIVER TYPES 5
JDBC URLS 7
EXECUTING DATABASE QUERIES 8
INSERTING AND UPDATING DATA 10
HANDLING DATES 11
CREATING OBJECTS USING DATABASE DATA 11
INSERTING OBJECTS INTO THE DATABASE 14
PREPARED STATEMENTS 15
TRANSACTIONS 16
STORED PROCEDURES 17
APPENDIX A: SETTING UP A WINDOWS DSN 20
APPENDIX B: CONNECTING TO AN HSQLDB DATABASE 21
APPENDIX C: CONNECTING TO AN ORACLE DATABASE 25

JP 31/05/2005

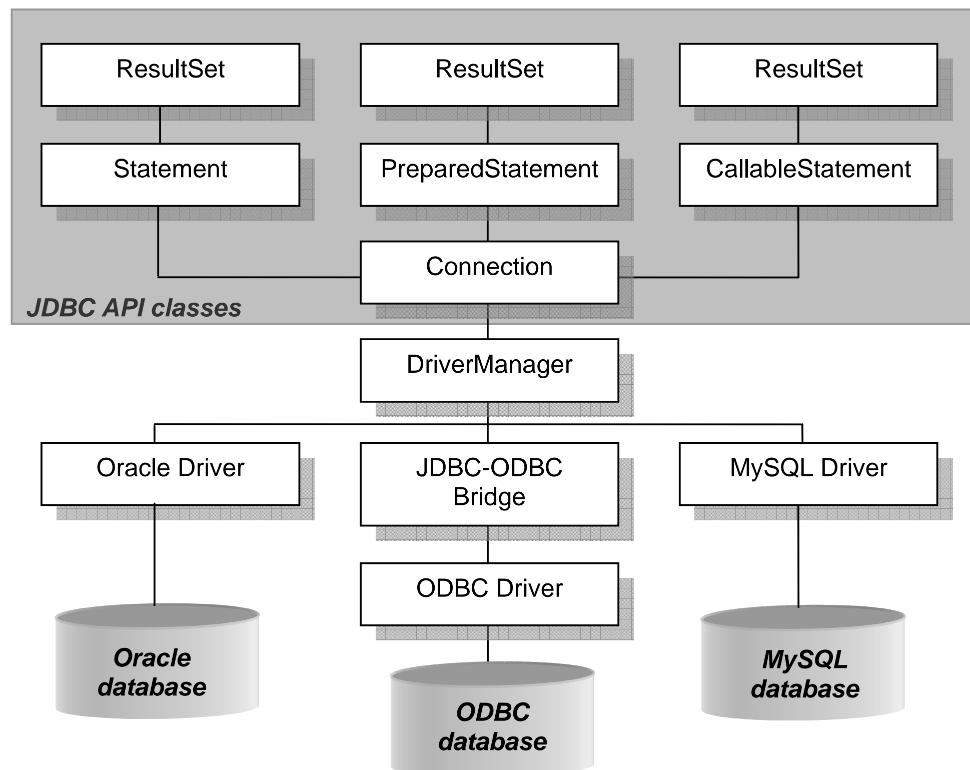
Introduction

The **JDBC API** provides Java applications with access to most database systems, using Structured Query Language (SQL). It is usually assumed that JDBC stands for **Java DataBase Connectivity**, although Sun claim officially that it doesn't!

Different database systems don't actually have much in common - basically just a similar purpose and a compatible query language. Every database has its own API to allow you to write programs that interact with the database. This means that writing code capable of interfacing with databases from more than one vendor is difficult. Cross-database APIs exist, most notably Microsoft's ODBC. ODBC is, however, limited to the Windows platform.

JDBC is Sun's attempt to create a cross-platform interface between databases and Java. With JDBC, you can count on a standard set of database access features and a particular subset of SQL. JDBC defines a set of interfaces that allow the Java programmer to access database functionality, including running queries and processing results. A database vendor or third-party developer writes a JDBC *driver*, which is a set of classes that implements these interfaces for a particular database system. To change the database used in a program, you simply need to change the driver which you use – the rest of the code stays the same.

The figure shows how an application uses JDBC to interact with one or more databases without knowing about the underlying driver implementations.



Note that Java classes which use the JDBC API classes must import them from the **java.sql** package, for example

```
import java.sql.*;
```

Creating a database connection

The following code shows a method which connects to a database and returns a ***Connection*** object. *Connection* is the JDBC API class which represents a database connection. The *Connection* returned by this method can then be used elsewhere in the program

```
/**
 * connection to database
 * @return a Connection
 */
public Connection dbConnect()
{
    Connection con = null;
    try{
        //Make sure the JdbcOdbcSriver class is loaded
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        //Try to connect to database
        con = DriverManager.getConnection("jdbc:odbc:Northwind");
    }
    catch (SQLException exc)
    {
        System.out.println("Error making JDBC connection: " +
                  exc.toString());
    }
     catch (ClassNotFoundException exc)
    {
        System.out.println("Error loading driver class: " +
                  exc.toString());
    }
    return con;
}
```

The key steps here are:

1. Loading the driver

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

This line loads the driver class and registers it with the JDBC DriverManager. This example loads a driver class called the JDBC-ODBC bridge driver.

2. Creating the connection

```
con = DriverManager.getConnection("jdbc:odbc:Northwind");
```

This line creates a connection object using a specific database name. In this case the database is a **Windows ODBC database** identified by the Windows **Data Source Name (DSN)** "Northwind". The examples in these notes use the Microsoft Access Northwind sample database, and assume that a DSN has been set up on your system for this database. See Appendix A for details on how to set this up.

Note that the checked exception *SQLException* must be handled when using JDBC API methods. The above example also handles *ClassNotFoundException*, so that any problems with loading the driver can be identified.

Closing the connection

Open connections should be closed before a program terminates. The following method closes a connection. It takes a *Connection* object as its parameter:

```
/**
 * close a database connection
 * @param con the database connection
 */
public void closeConnection(Connection con)
{
     try {
        if (!con.isClosed())
        {
            con.close();
        }
      }
    catch (SQLException exc)
    {
        System.out.println("Error closing JDBC connection: " +
            exc.toString());
    }
}
```

JDBC Driver types

There are JDBC drivers for most databases. There are **FOUR** categories of drivers:

Type 1 JDBC-ODBC bridge drivers

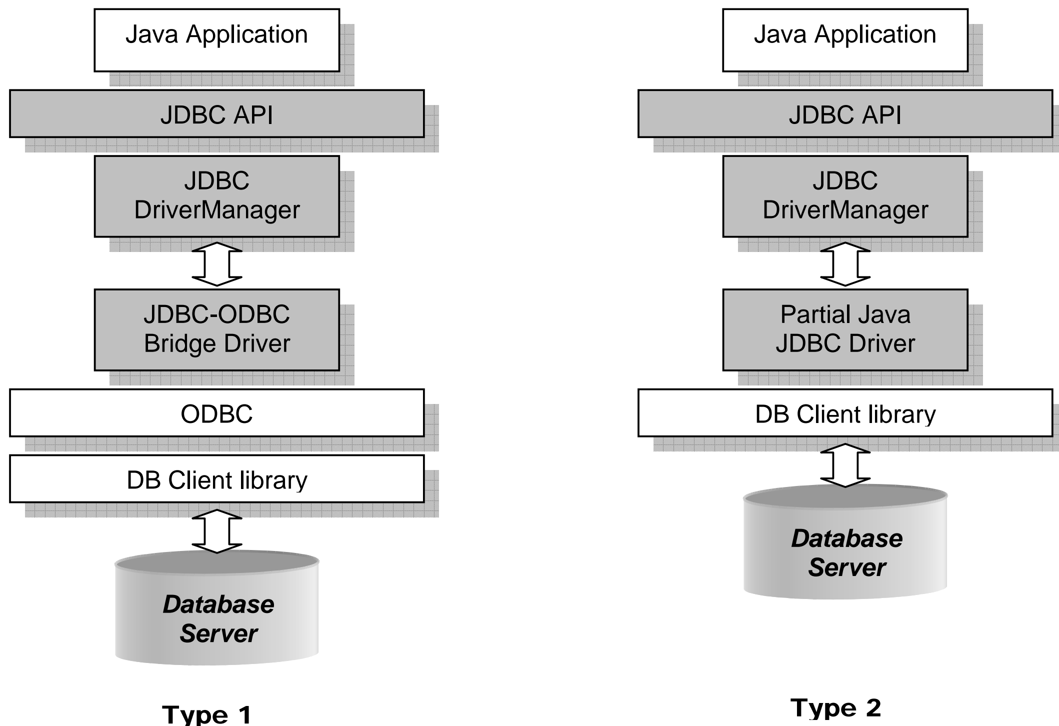
Type 1 drivers use a bridge technology to connect a Java client to an ODBC database system, usually on a Windows platform. The Sun JDBC-ODBC Bridge is the only existing example of a Type 1 driver. Type 1 drivers require non-Java software to be installed on the machine running your code, and they are implemented using native code (i.e. code which is specific to the operating system running the database – not Java code).

Type 2 Native-API partly Java drivers

Type 2 JDBC drivers talk directly to the API of the DBMS, rather than any mapping layer such as ODBC, and can be accessed from Java.

This implies that the driver either does not provide the complete JDBC API (but provides enough to drive the native database API) or is not written completely in Java (thus losing out in cross-platform functionality).

Technically, this type of driver is the most efficient user of machine resources, but this advantage is far outweighed by the need to write a different JDBC driver for each DBMS and for each platform.

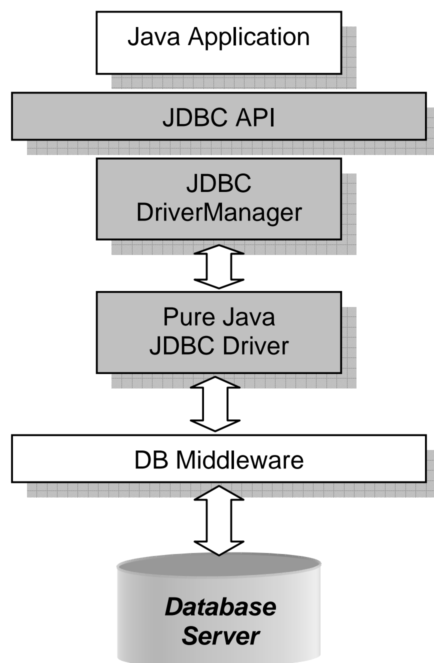


Type 3 Net-protocol All-Java drivers

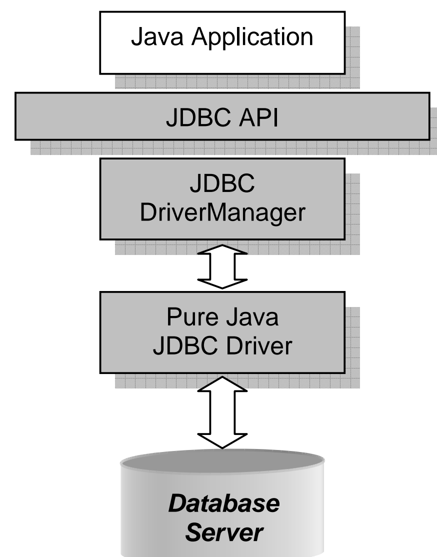
These drivers use a networking protocol and middleware to communicate with a server. The server then translates the protocol to DBMS specific function calls. Type 3 JDBC drivers are the most flexible JDBC solution They do not require any native binary code on the client. A Type 3 driver does not need any client installation. These drivers are especially useful for applet deployment, since the actual JDBC classes can be written entirely in Java and downloaded by the client on the fly.

Type 4 Native-protocol All-Java drivers

Type 4 drivers are written entirely in Java. They understand database-specific networking protocols and can access the database directly without any additional software. This means that there is no client installation or configuration. However, a Type 4 driver may not be suitable for some applications if the underlying protocol doesn't handle issues such as security and network connectivity well.



Type 3



Type 4

JDBC URLs

A JDBC driver uses a JDBC URL to identify and connect to a specific database. JDBC URLs are of the form:

```
jdbc:driver:databasename
```

In the above example the JDBC-ODBC Bridge driver was used to connect to an ODBC database with the DSN *"Northwind"*. No username or password needed to be specified for this Access database:

```
con = DriverManager.getConnection("jdbc:odbc:Northwind");
```

The following example would be used to connect to an Oracle 9i database called *"oracle9a"* on a computer named *"MyServer"*, and supplies a username (Scott) and password (Tiger):

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

con = DriverManager.getConnection
 ("jdbc:oracle:thin:@MyServer:1521:oracle9a","scott", "tiger");
```

This example uses the Oracle Thin JDBC driver, which is a Type 4 driver. See Appendix B for more information on connecting to Oracle.

Executing database queries

Once you have created a *Connection* you can use it to execute SQL statements. There are actually three kinds of statement classes in JDBC:

Statement
Represents a basic SQL statement

PreparedStatement
Represents a precompiled SQL statement, which can offer improved performance

CallableStatement
Allows JDBC programs complete access to stored procedures within the database.

The simplest to use is the *Statement* class. A *Statement* object is created by the *createStatement* method of a *Connection*.

The *Statement* can then be used to execute queries using its *executeQuery* method. This method returns a *ResultSet* object which contains the results of the query.

The following example shows a method which creates a *Statement*, executes a query and displays the results on the console. It takes a *Connection* as a parameter – this assumes that the dbConnect method listed above has been run to create the connection to the Northwind database.

```
/**
 * performs a query and displays recordSet
 */
public void queryToRecordSet(Connection con)
{
    Statement stmt = null;
    try{
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * from SHIPPERS");

        // display the results
        System.out.println("Results of query to recordset");
        while(rs.next())
        {
          System.out.println(rs.getInt(1) + ", " + rs.getString(2) + ",
                  " + rs.getString(3));
        }
    }
    catch (SQLException exc)
    {
        System.out.println("Error performing query: " + exc.toString());
    }
}
```

ResultSet methods include:

- **first**()
- **last()**
- **next**()
- **previous()**
- **getString**(String *columnname*) or **getString**(int columnNumber)
- **getInt**(String *columnname*) or **getInt**(int columnNumber)
- **getObject**(String *columnname*) or **getObject**(int columnNumber)

Putting it together

The following example shows a main method which calls the methods listed above to connect to the Northwind database and display the results of a query. The database methods are in a class called *JdbcDemo*.

```
public static void main(String[] args) {
    JdbcDemo jdbcDemo1 = new JdbcDemo();

    // open connection
    Connection conn = jdbcDemo1.dbConnect();

    // simple query
    jdbcDemo1.queryToRecordSet(conn);

    // close connection
    jdbcDemo1.closeConnection(conn);
  }
```

The output from this code looks like this:

```
Results of query to recordset
1, Speedy Express, (503) 555-9831
2, United Package, (503) 555-3199
3, Federal Shipping, (503) 555-9931
```

Inserting and updating data

Not all SQL statements return results. INSERT, UPDATE and DELETE statements simply alter the contents of the database. The *Statement* class has an *executeUpdate* method which does not return a *ResultSet*. In fact it returns an integer which indicates the number of rows in the database which were altered.

The following method demonstrates inserting and updating data. Again, a *Connection* object is required as a parameter. Note that this method does not display any results – the *queryToRecordSet* method could be used to display the contents of the table after updating.

```
/**
 * updates and inserts data
 * @param con a database connection
 */
public void insertAndUpdateRecord(Connection con)
{
    Statement stmt = null;
    try{
        stmt = con.createStatement();

        String updateQuery = "UPDATE SHIPPERS SET CompanyName = 'Speedy
            Inc' WHERE ShipperID = 1";
        stmt.executeUpdate(updateQuery);

        String insertQuery = "INSERT INTO SHIPPERS (CompanyName, Phone)
            VALUES ('Bell International', '(01698) 283100')";
        stmt.executeUpdate(insertQuery);
    }
    catch (SQLException exc)
    {
        System.out.println("Error performing query: " + exc.toString());
    }
}
```

Note that the INSERT statement used here does not include a value for the first field in the table, *ShipperID*, as this is an autonumber field in the Access database.

Handling dates

JDBC defines three classes devoted to storing date and time information: *java.sql.Date*, *java.sql.Time*, and *java.sql.Timestamp*. These correspond to the SQL DATE, TIME, and TIMESTAMP types. The usual *java.util.Date* class is not suitable for any of them.

The SQL DATE type contains only a date, so the *java.sql.Date* class contains only a day, month, and year. SQL TIME (*java.sql.Time*) includes only a time of day, without date information. SQL TIMESTAMP (*java.sql.Timestamp*) includes both.

Different DBMS packages have different methods of encoding date and time information. JDBC supports the ISO date escape sequences, and individual drivers must translate these sequences into whatever form the underlying DBMS requires. The syntax is:

```
{d 'yyyy-mm-dd'}
{t 'hh:mm:ss'}
{ts 'yyyy-mm-dd hh:mm:ss.ms.microseconds.ns'}
```

Here is an example that uses a date escape sequence:

```
stmt.executeUpdate("INSERT INTO FRIENDS(BIRTHDAY) VALUES ({d '1978-12-
14'})")
```

Creating Objects using database data

So far all we have done with our data is to display it on the console. In object-oriented programs it is often necessary to use the data to create objects which can be used elsewhere in the program.

The next example shows how the contents of the Shippers table can be stored in an *ArrayList* data structure object. This example assumes that a *Shipper* class has been defined. A *Shipper* object has attributes which correspond to the fields in the database table, and can represent a single row in the table. We say that the Shipper class **models the data**.

```
public class Shipper {
    public int shipperId;
    public String companyName;
    public String phone;

  // default constructor
  public Shipper(){
        shipperId = 0;
        companyName = "None";
        phone = "None";
  }

  // constructor which sets all attribute values
  public Shipper(int id, String name, String ph){
        shipperId = id;
        companyName = name;
        phone = ph;
  }
}
```

The method which performs the query is as follows. As usual, a *Connection* object is required as a parameter.

```
/**
 * queries database and stores the results as objects for use elsewhere
 * in an application
 * @param con the database connection
 * @return a List of Shipper objects
 */
public List queryToObjects(Connection con)
{
    List shippers = new ArrayList();
    Shipper ship = null;
    Statement stmt = null;

    try{
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * from SHIPPERS");

        /* add each database row to List of shippers*/
        while (rs.next())
        {
            // construct new Shipper object from RecordSet row
            ship = new Shipper(rs.getInt(1),rs.getString(2),
                  rs.getString(3));
            shippers.add(ship);
        }
        System.out.println("Data stored as a List of Shipper objects");
    }
    catch (SQLException exc)
    {
        System.out.println("Error performing query: " + exc.toString());
    }

    return shippers;
}
```

Note that this method returns an *ArrayList* object (actually the reference type here is *List*, but the runtime type will be *ArrayList*) which contains the data in the form of *Shipper* objects. It does not display any results. The objects in the *ArrayList* can be passed to any other class or method in the program for display or processing.

The following method demonstrates the use of an Iterator to display the contents of the *ArrayList*. This method takes the *ArrayList* as a parameter.

```
/**
 * displays the contents of a list of Shipper objects
 * @param shippers
 */
public void displayShippers(List shippers)
{
        Shipper ship = null;

        System.out.println("Contents of list of Shippers");
        for (Iterator itr = shippers.iterator() ; itr.hasNext() ;) {
          ship = (Shipper) itr.next();
          System.out.println(ship.shipperId + ", "+  ship.companyName +
                  ", "+  ship.phone);
        }
}
```

This method does not access the database – it simply displays previously created objects.

The following code in the main method will call the above methods to query the database and display the results. Although these are shown together as a demonstration, in a real application method calls like these could be in completely different parts of the program, with the database data passed using a single object.

```
List shipperList = jdbcDemo1.queryToObjectsDemo(conn);
jdbcDemo1.displayShippers(shipperList);
```

Inserting Objects into the database

It is often necessary in object-oriented programs to store objects created in the program in a database table. The following example method takes a Shipper object as a parameter, and uses its attributes to inserts a new row into the Shippers table. The first part of the SQL query is created as a StringBuffer, and the Shipper values are appended to create a complete query string.

```
/**
 * insert a Shipper object as a new row in the Shippers table
 * @param con the database connection
 * @param ship a Shipper object
 */
public void insertNewObject(Connection con, Shipper ship)
{
      Statement stmt = null;
      try{
          stmt = con.createStatement();

          // create query string
          StringBuffer insertQuery = new StringBuffer("INSERT INTO
                  SHIPPERS (CompanyName, Phone) VALUES ('");
          insertQuery.append(ship.companyName);
          insertQuery.append("', '");
          insertQuery.append(ship.phone);
          insertQuery.append("')");

          stmt.executeUpdate(insertQuery.toString());

      }
      catch (SQLException exc)
      {
          System.out.println("Error performing query: " +
                  exc.toString());
      }
}
```

This method could be called like this:

```
Shipper ship = new Shipper(0, "Rapid Transit", "(01698)283100");
jdbcDemo1.insertNewObject(conn, ship);
```

Prepared statements

The *PreparedStatement* class is closely related to the *Statement* class. They can both be used to execute SQL statements. The difference is that *PreparedStatement* allows you to precompile your SQL and run it repeatedly. You can adjust specified parameters each time the SQL is run – for example you could repeatedly run an INSERT statement with different values to be inserted each time.

The main advantage of using a *PreparedStatement* is that you can significantly **improve performanc**e by doing compilation once only, as SQL string processing is a large part of the work done by the database.

The following method shows the use of a *PreparedStatement* to insert two database rows. In a real program, the *PreparedStatement* would probably be re-used many more times than this.

```
/**
 * performs a query using a prepared statement
 * @param con the database connection
 */
public void preparedStatementQuery(Connection con)
{
    PreparedStatement pstmt = null;
    try{
        String insertQuery = "INSERT INTO SHIPPERS (CompanyName, Phone)
            VALUES (?,?)";
        pstmt = con.prepareStatement(insertQuery);

        pstmt.setString(1, "DHM Couriers");
        pstmt.setString(2, "(503) 555-1139");
        pstmt.executeUpdate();

        pstmt.setString(1, "Mercury Inc.");
        pstmt.setString(2, "(503) 555-1133");
        pstmt.executeUpdate();

    }
    catch (SQLException exc)
    {
        System.out.println("Error performing query: " + exc.toString());
    }
}
```

The *PreparedStatement* is created using the *prepareStatement* method of the *Connection* object, which takes a query string as a parameter. The query string can contain **?** symbols which indicate parameters to be supplied at runtime.

Like a *Statement*, the *PreparedStatement* object has *executeUpdate* (and *executeQuery*) methods. It also has methods, such as *setString* used in the example, to set the values of parameters. For example, the line:

```
pstmt.setString(1, "DHM Couriers");
```

causes the first **?** in the query string to replaced with “DHM Couriers” when the statement is executed.

Transactions

A transaction is a group of several operations that behave atomically – i.e. as if they are a single operation. Executing a transaction involves the following steps:

- Start the transaction
- Perform its component operations
- Commit if all operations succeeded **OR** Rollback if one operation failed

Rolling back returns the database to the state before the start of the transaction, as if none of the operations had taken place.

Transactions can be important in many database situations. For example, if a stock management system moves stock items from a STOCK table to a SHIPPING table, then using transactions can prevent the possibility of an item being deleted from STOCK and not being added successfully to SHIPPING.

To **start a transaction** you call the *setAutoCommit* method of the *Connection* with a parameter value of "false". This means that the transaction needs to be committed manually using the commit method of the *Connection*, e.g.

```
con.setAutoCommit(false);
```

A set of **component operations** are then executed, e.g.

```
String updateQuery = "UPDATE SHIPPERS SET CompanyName = 'Package
      Force' WHERE ShipperID = 1";
stmt.executeUpdate(updateQuery);

String insertQuery = "INSERT INTO SHIPERS (CompanyName, Phone)
      VALUES ('Bell International', '(01698) 283100')")";
stmt.executeUpdate(insertQuery);
```

Note that the second query string here has an error – it refers to the SHIPERS table rather than the SHIPPERS table. This will cause an SQLException to be thrown as the table will not be found.

Finally, the **transaction is committed**:

```
con.commit();
```

If an *SQLException* is thrown by any operation, the transaction is **rolled back** in the *catch* block:

```
con.rollback();
```

The following method performs two actions on the Shippers table as a transaction. IF the code is run as shown, the incorrect table name in the second query will cause an SQLException, and the database will not be changed. If this error is corrected and the code run again, both queries will be executed and the database will be altered.

```
/**
 * performs a transaction
 * @param con the database connection
 */
public void transactionsDemo(Connection con)
{
  Statement stmt = null;
    try{
        con.setAutoCommit(false);
        con.setTransactionIsolation(con.TRANSACTION_READ_COMMITTED);
        stmt = con.createStatement();

        // perform a series of actions as a transaction
        String updateQuery = "UPDATE SHIPPERS SET CompanyName = 'Package
            Force' WHERE ShipperID = 1";
        stmt.executeUpdate(updateQuery);

        String insertQuery = "INSERT INTO SHIPERS (CompanyName, Phone)
            VALUES   ('Bell   International',   '(01698)   283100)')";
        stmt.executeUpdate(insertQuery);

        // commit transaction
        con.commit();
        con.setAutoCommit(true);  //Access gives Invalid Transaction
                                  //State error on closing without this
    }
    catch (SQLException exc)
    {
        System.out.println("Error performing query, rolling back: ");
        System.out.println(exc.getMessage());
        try{
            con.rollback();
            con.setAutoCommit(true);
        }
        catch (SQLException trexc)
        {
            System.out.println("Error rolling back transaction: "
                  + trexc.toString());
         }
      }
}
```

Note that when using Access it was necessary to set *AutoCommit* back to “true” after either committing or rolling back, otherwise Access gave an error on closing the connection. Note also that *rollback* throws *SQLException*, and in the example this is handled inside the main *catch* block.

Stored Procedures

Most DBMS systems include some sort of internal programming language (e.g., Oracle's PL/SQL). These languages allow database developers to embed application code directly within the database and then call that code from other applications. The advantage of this approach is that the code can be written just once and then used in multiple different applications. It also allows application code to be divorced from the underlying table structure. If stored procedures

handle all of the SQL, and applications just call the procedures, only the stored procedures need to be modified if the table structure is changed later on.

JDBC allows you to call a database stored procedure from a Java application. The first step is to create a *CallableStatement* object, in a similar way to creating *Statement* and *PreparedStatement* objects. A *CallableStatement* object contains a call to a stored procedure; it does not contain the stored procedure itself. The first line of code below creates a call to the stored procedure SHOW_SHIPPERS using the *Connection* con. When the driver encounters "{call SHOW_SUPPLIERS}" , it will translate into the native SQL used by the database to call the stored procedure named SHOW_SUPPLIERS .

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

EXERCISE

Part A
In this of this exercise you can make use of the code given in these notes.

1. Create a Netbeans project called *jdbcexercise* and add a new class *JDBCExercise*. This class should contain methods required to connect to the Northwind database and display the contents of the Shippers table. Create a *main* method to execute this operation.

2. Modify your *JDBCExercise* class so that it updates one record in the Shippers table, and inserts a new record, and then displays the new contents of the table. When modifying for these exercises, you can simply add a new method and use comments to choose which methods are called when the main method is run, e.g.

```
public static void main(String[] args) {
      JDBCExercise jdbcEx1 = new JDBCExercise();

      Connection conn = jdbcEx1.dbConnect();

      // simple query – commented out
      // jdbcEx1.queryToRecordSet(conn);

      // insert and update - called
      jdbcEx1.insertAndUpdateRecord(conn);

      jdbcEx1.closeConnection(conn);
}
```

3. Modify your *JDBCExercise* class so that it queries the database and stores the contents of the Shippers table in an *ArrayList* of *Shipper* objects, and uses a separate method to display the contents of the *ArrayList*.

4. Modify your *JDBCExercise* class so that it creates a new *Shipper* object and uses it to insert a new database row.

5. Modify your *JDBCExercise* class so that it uses a prepared statement to insert two new rows in the Shippers table.

6. Modify your *JDBCExercise* class so that it uses a transaction to execute two queries. Check that the transaction rolls back if there is an error, and commits otherwise.

Part B
Add a new class *JDBCExerciseB* to your project. Create and test methods which do the following:

1. Display the CompanyName, ContactName and City from the Customers table using a RecordSet.

2. Display the contents of the OrderDetails table which belong to Order ID 10248. You should create an OrderDetail class and store the data in a List of OrderDetail objects. You should then display the contents of the List.

3. Create a new OrderDetail object to belong to Order ID 10248, and use it to add a row to the OrderDetails table

Appendix A: Setting up a Windows DSN

An Access database can be configured in Windows as an **ODBC Data Source**. Once an ODBC Data Source is configured, a Java application can connect to it using the JDBC-ODBC Bridge driver.

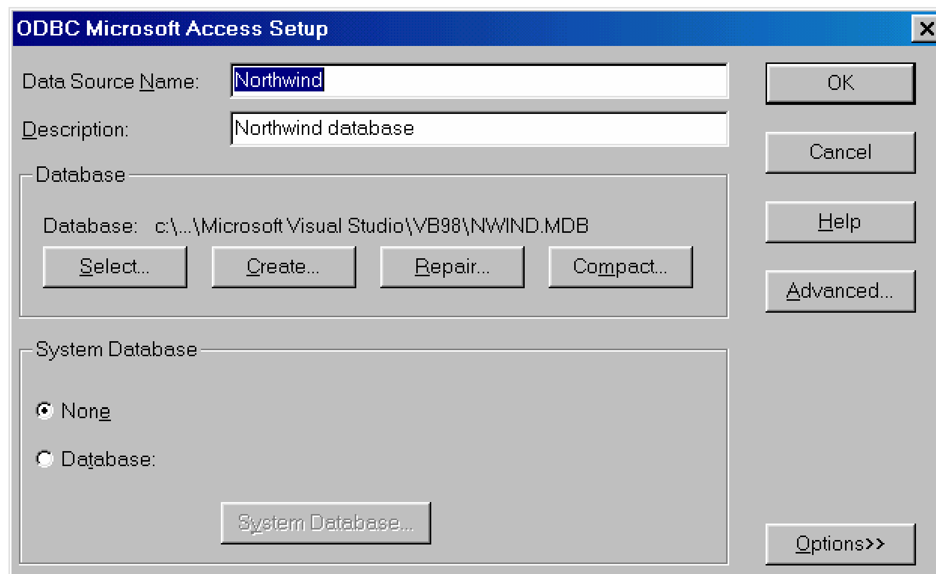
Many other kinds of databases which run on Windows can also be configured in the same way, so Java can connect in the same way to just about any kind of Windows database, e.g. Microsoft SQL Server, Oracle, Interbase, etc.

Note that ODBC may not be the best way to connect to a particular database. For example, although you can set up an Oracle database as an ODBC Data Source, it is usually better to use a specific Oracle driver.

If the database has not been configured, then you will need to set up an ODBC Data Source Name. Open the **32-bit ODBC Control Panel** in Windows 98 (it's under **Control>Panel Administrative Tools** in Windows 2000/XP). Select the **System DSN** tab. This allows you define a Data Source Name which is available to all users.

To set up the Access Northwind database, click **Add** and select the **Microsoft Access Driver (*.mdb)**, and then click **Finish**.

The ODBC Microsoft Access Setup window appears. Fill in the DSN 'Northwind' and the description 'Northwind Database'. Click **Select** and select your 'nwind.mdb' or 'northwind.mdb' databse in the file dialog box which opens. (You will need to know where the file is located). Click **OK**. You will then see Northwind in the System DSN tab. Click **OK** here to finish.



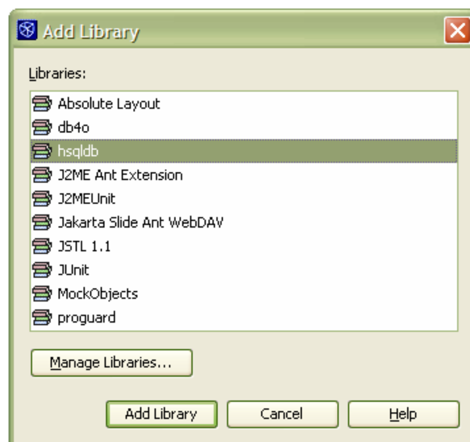
If your PC has its Control Panel disabled, you may be able to display the above dialog using the **32-bit Administrator** option in on of the **Oracle** menus under **Start>Programs**.

Appendix B: Connecting to an HSQLDB Database

HSQLDB is a SQL relational database engine written in Java. It has a JDBC driver and supports a offers a small, fast database engine which offers both in-memory and disk-based tables. It can be used in the examples in these notes as an alternative to Access. It is simple to set up – no DSN is required, and the database engine can be started and managed within NetBeans. It also supports some features which are commonly found in databases, but which Access lacks.

HSQLDB Library

In order to use HSQLDB in a NetBeans project, you need to add a suitable library to the project. You may find that the NetBeans installation you are using has the library included. Right-click on the *Libraries* node in your project and select **Add Library…**. If *hsqldb* appears in the **Add Library** dialogue then you can simply click the **Add Library** button.



If not, you need to tell the project to use the file ***hsqldb.jar*** which you can download from you course web site. You can simply add this file to the project as a JAR. You can also use the **Manage Libraries** option. This opens the **Library Manager** and allows you to create a new library, called *hsqldb,* and add the JAR file to the new library. You can then add the new library to your project.

The Libraries node in the project should now contain the *hsqldb* library.

Using HSQLDB for the JDBC exercise

The following notes will help you modify the examples in these notes to use an HSQLDB database instead of Access. The same modifications can be used for examples which you will come across later in your course.

Before you connect to database, you need to start the database server and specify where the database files will be kept. You can do this by adding the following class to your project, and running the file:

```
package jdbcdemo;

import org.hsqldb.Server;
import java.sql.*;
import java.util.*;

public class StartHsqlServer {

    public static void main(String[] args){
        String  serverProps;
        String  url;
        String  user     = "sa";
        String  password = "";
        Server  server;

        try{
            serverProps = "database.0=file:e:/hsqldatabases/northwind";
            url         = "jdbc:hsqldb:hsql://localhost";
            server      = new Server();

            server.putPropertiesFromString(serverProps);
            server.setLogWriter(null);
            server.setErrWriter(null);
            server.start();
        }
        catch (Exception e){
            System.out.println("Error starting server: " +
                  e.toString());
        }
    }
}
```

The *serverProps* variable in this example specifies that the database files will be stored in a folder *e:/hsqldatabases*, and will start with the name *northwind* (*northwind.log*, *northwind.script*, *northwind.properties*). You can back up the database simply by taking a copy of these files.

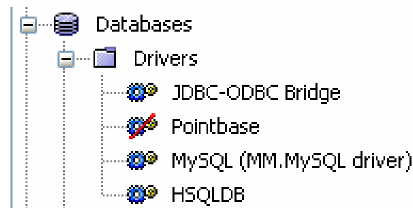
The code to connect to the database is as before, but the driver and URL showuld be changed to the following:

```
Class.forName("org.hsqldb.jdbcDriver");
con = DriverManager.getConnection("jdbc:hsqldb:hsql://localhost");
```

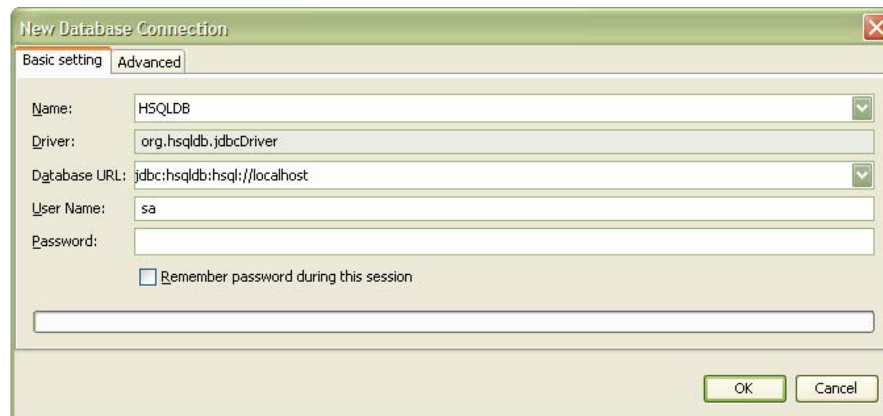
Run the file *StartHsqlServer* to start the database server. You can now run *JdbcExercise* – this will connect to the database, but will not work correctly as you have not yet added any data to the database.

Managing the HSQL database

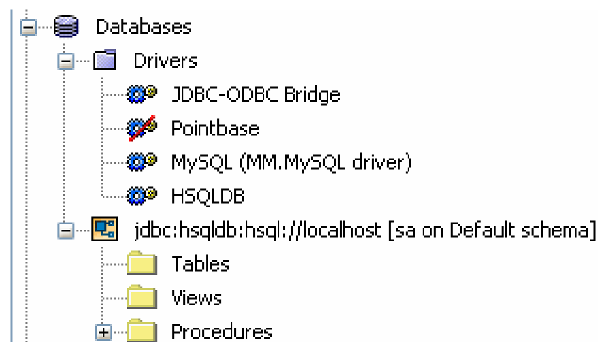
The HSQL database can be managed from the **Runtime** window in NetBeans. Open the databases node and open the *Drivers* folder. The *HSQLDB* driver may be there already – if not, right-click on the *Drivers* folder and select **Add Driver…**, and choose the same *hsqldb.jar* file as before.



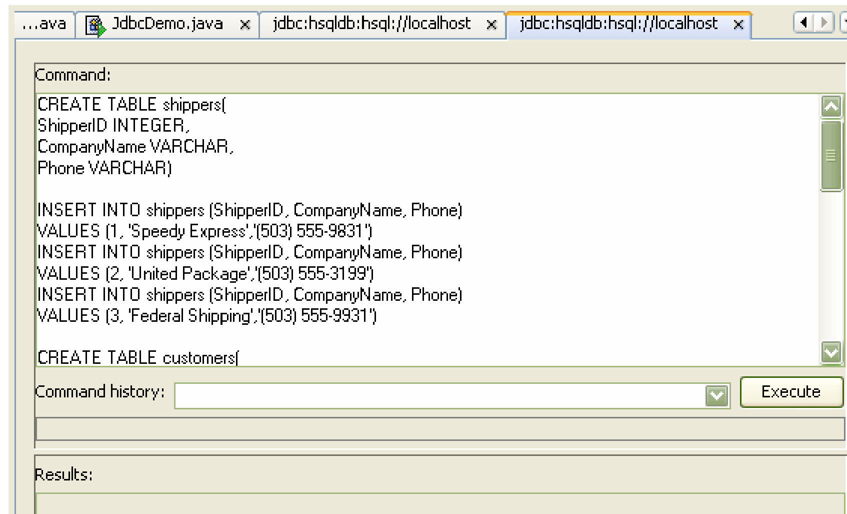
Now you need to connect to the database. Right-click on the HSQLDB driver and select Connect Using… . Use the Database URL *jdbc:hsqldb:hsql://localhost* and username *sa*.



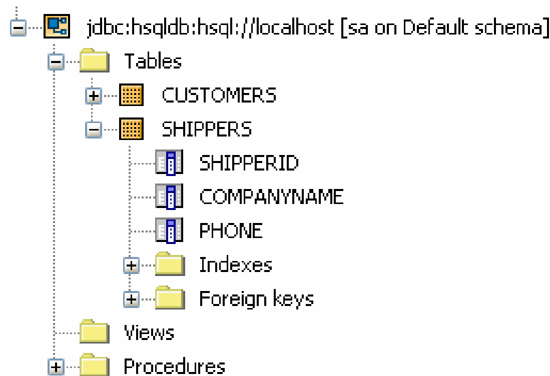
A new connection is added in the **Runtime** window:



You need to add some data. Right-click on the connection in the **Runtime** window and select **Execute Command…**. Paste the contents of the file *northwind.sql*, which you can download from your course web site, into the **Command** box, and click the **Execute** button.



Two tables should be added to the database. You can now view the data or execute any other SQL commands.



Running JdbcExercise

You can now return to the **Project** window and run the file *JdbcExercise*, which should connect to the database and return the correct results.

Note that the *insert* code will need to be modified to set a value for *ShipperID*, as this is not an autonumber field in the HSQLDB database. To do items 2&3 in Part B of the exercise, you will need to add new tables and data to the HSQL database.

Shutting down

To shut down the database, enter the command SHUTDOWN in the **Command** box for this database connection.

Appendix C: Connecting to an Oracle Database

In order to connect to Oracle you will need a suitable JDBC driver for the versions of Oracle and Java which you are using. Drivers can usually be downloaded from Oracle's web site.

The driver is supplied in the form of a Zip file. The example code below was used to connect to an Oracle 9i database called **oracle9a**.

The driver used was version 9.0.1.4 for JDK1.2 and 1.3. The driver file is called **classes12.zip**. This file was added as a required library in JBuilder using **Project>Properties**.

The JDBC URL needs to specify the following:

Driver:	jdbc.oracle.thin
Server Name:	the name or IP address of the computer hosting the database – *myserver* in this case
Port:	usually 1521 if Oracle was installed with default values
Database name (SID):	*oracle9a* in this case
Username:	using built-in account Scott for testing
Password:	Tiger

The dbConnect method code is:

```
/**
 * connection to database
 * @return a Connection
 */
public Connection dbConnect()
{
    Connection con = null;
    try{
        //Make sure the JdbcOdbcSriver class is loaded
        Class.forName("oracle.jdbc.OracleDriver");

        //Try to connect to database
        con =
            DriverManager.getConnection("jdbc:oracle:thin:
            @myServer:1521:oracle9a","scott", "tiger");
    }
    catch (SQLException exc)
    {
        System.out.println("Error making JDBC connection: " +
            exc.toString());
    }
     catch (ClassNotFoundException exc)
    {
        System.out.println("Error loading driver class: " +
            exc.toString());
    }
    //Return true if successful
    return con;
}
```