SCHOOL OF COMPUTER SCIENCE

PRIFYSGOL BANGOR UNIVERSITY

Java Technologies Mini-Project 1

Introducing Maven

Dave Perkins

This lab was tested on NetBeans 8.2

Introduction

This first mini-project is intended to introduce students to the use of a project management tool known as Maven. Although this tool can be used in conjunction with either Eclipse or NetBeans it is recommended that you use the latter IDE as the laboratory notes assume that the development environment is NetBeans. In this mini-project you will:

- create a Maven project based on a simple "Hello World!" application;
- create a Maven project for a simple bank account program;
- explore the use of the NetBeans "Matisse" GUI builder;
- explore Maven's unit test facilities;
- create project dependencies;
- use a POM file to manage project resources;
- work with a light-weight database such as hsql.

You will also be directed to various readings which are intended to deepen your background knowledge of this tool.

Reading

Part of the work associated with this mini-project consists of reading. The following sites and /or documents should be consulted:

- A useful beginners tutorial for Maven can be found at

  http://tutorials.jenkov.com/maven/maven-tutorial.html

- Also try some or all of the following for further info on Maven

  http://en.wikipedia.org/wiki/Apache_Maven
  http://www.javaworld.com/article/2072203/build-ci-sdlc/an-introduction-to-maven-2.html
  http://books.sonatype.com/mvnex-book/reference/public-book.html
  http://www.avajava.com/tutorials/categories/maven

- For a brief introduction to the light-weight database **hsql** consult Appendix 1. The official **hsql** site has a lot of useful information:

  http://hsqldb.org/

What is Maven?

According to the Apache Software Foundation

> *Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.*[1]

Maven is widely used in industry and the kind of support it provides is indispensable for the development of large software projects.

Maven is a project management and comprehension tool and provides developers with a complete *build lifecycle framework*. A development team can automate the project's build infrastructure in almost no time as Maven uses a standard directory layout and a default build lifecycle. In case of multiple development teams environment, Maven can set-up the way to work as per standards in a very short time. As most of the project setups are simple and reusable, Maven makes life of developer easy while creating reports, checks, build and testing automation setups.

Maven provides developers with the facility to manage the following:

- Builds
- Documentation
- Reporting
- Dependencies
- SCMs
- Releases
- Distribution
- Mailing lists

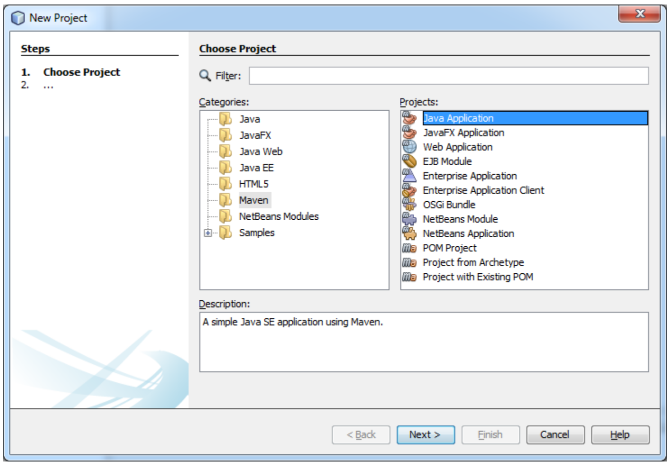
To summarize, Maven simplifies and standardizes the project build process: it handles compilation, distribution, documentation, team collaboration and other tasks seamlessly, increases reusability and takes care of most of build related tasks.

We shall be using Maven as an integrated component of NetBeans but it should be noted that Maven can be used as a stand-alone to manage the process of software development.

[1] See http://maven.apache.org/ for more information.

Exercise 1: Creating a Maven Project Using NetBeans (5%)

This first exercise involves the creation of a simple Java application using Maven support. To get started click on **File → New Project**. You should see the following screen:

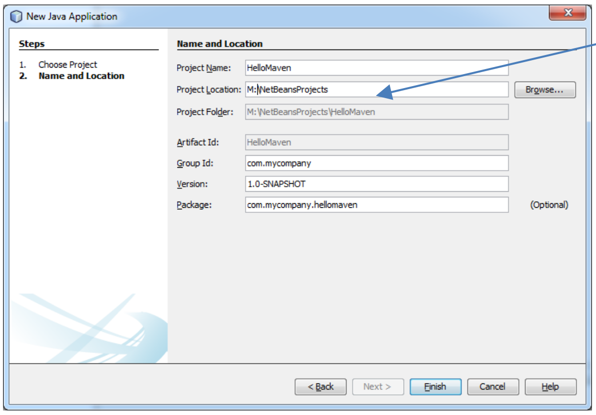


Description: A simple Java SE application using Maven.

Figure 1 Creating a Maven Project

- Under **Categories** select **Maven**
- Under **Projects** select **Java Application**
- Click **Next**

The following dialog then asks you to supply a name for your Maven project as well as its location. These are the only two values required, so once you have decided where to store the project on disc click **Finish**.



Project Name: HelloMaven
Project Location: M:\NetBeansProjects
Project Folder: M:\NetBeansProjects\HelloMaven
Artifact Id: HelloMaven
Group Id: com.mycompany
Version: 1.0-SNAPSHOT
Package: com.mycompany.hellomaven

Make sure you use your M-drive (M:)

Figure 2 Project name and location

At this point Maven will create what is known as the *standard project structure*. Once this structure has been created you will able to develop and run programs using the facilities provided by the project.

Figure 3 below show the Maven project which we have just built, but without the App.java.

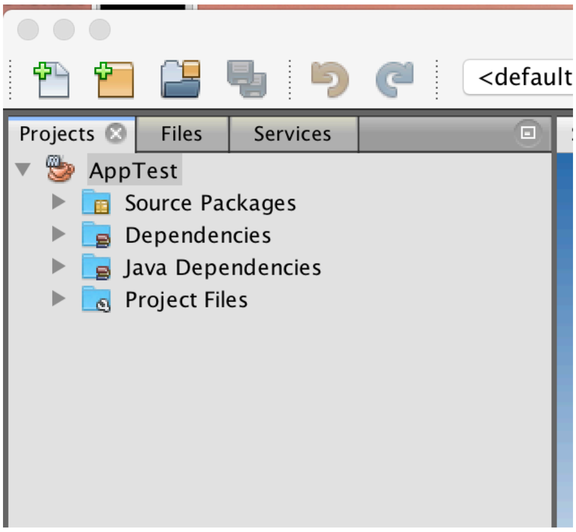
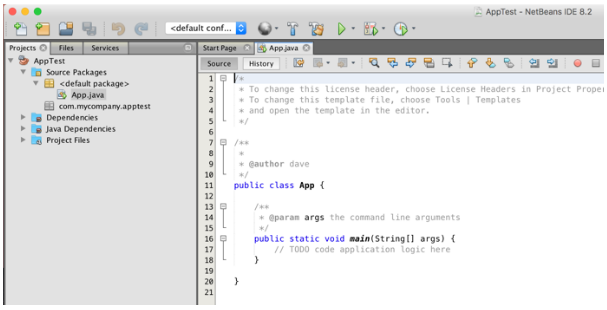


Figure 3 Maven Project Structure

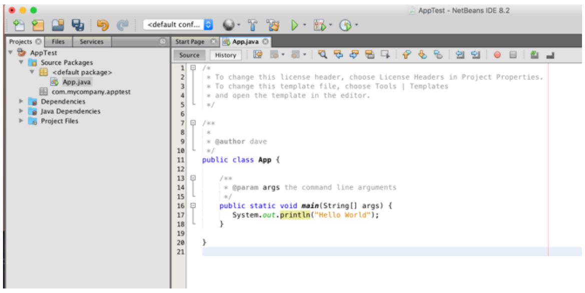
You will need to add a Java Main Class (App.java) to the Project, then in the properties (run) of the project make is a default Java application.



```java
/*
 * To change this license header, choose License Headers in Project Proper
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author dave
 */
public class App {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

Figure 3.1: Empty main class

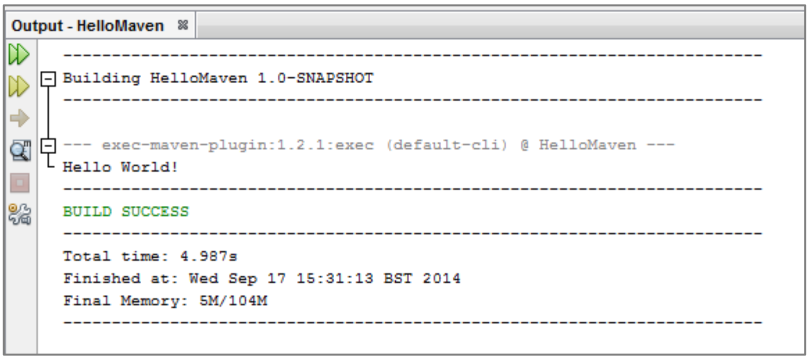
Add some code to this App.java to make a version of the well known “Hello World!” program. Bring the file **App.java** into the editor pane and then *right click* to access the *context menu*.



```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author dave
 */
public class App {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
       System.out.println("Hello World");
    }

}
```

Figure 3.2: Hello World code added

Select the menu item **Run** or press the green Play button in the menu bar ▷. If everything has been set up correctly Maven should generate the famous (or perhaps infamous) message “Hello World!” in the output pane. To run the program again simply click on the green arrows on the left-hand side of **Output** window – Figure 4.



```
------------------------------------------------------------------------
Building HelloMaven 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ HelloMaven ---
Hello World!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 4.987s
Finished at: Wed Sep 17 15:31:13 BST 2014
Final Memory: 5M/104M
------------------------------------------------------------------------
```

Figure 4 Viewing Program Output

Before moving on to the next exercise spend a little time exploring the contents of the other components of the project structure. Make sure you can locate the following files:

- `pom.xml`
- `rt.jar`
- `App.java`

Exam the contents of these files and try to guess their purpose. Also use the **File** view in NetBeans to study the *standard directory structure* which Maven creates.

Exercise 2: Using Project Facilities (5%)

When a Maven project is created a number of folders are automatically generated inside the project. The Test Packages should in theory appear, however in the latest version of Netbeans (8.2) this is not the case. You can force NetBeans to create the Test Packages folder going to Tools>Create/Update Test. In this exercise we explore the contents of the folder named **TestPackages**.

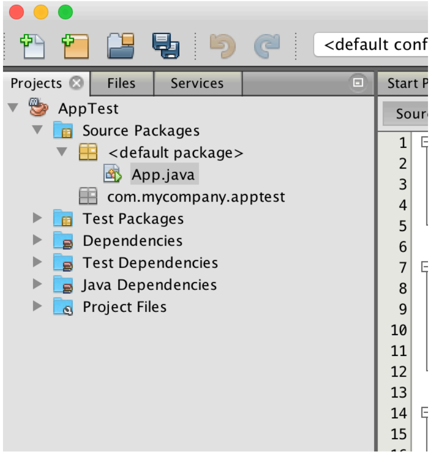
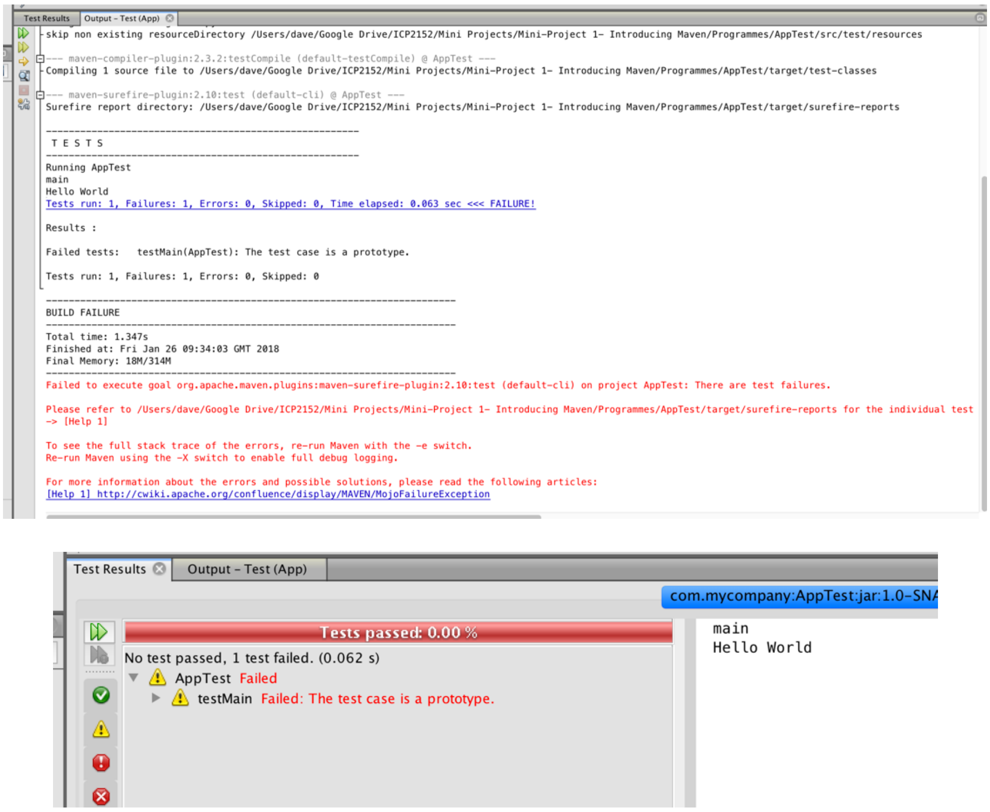


Figure 5 Contents of HelloMaven Project

Open the file **AppTest.java** in the edit pane and examine the contents. This is an automatically generated test program and can be used to test the application located in **Source Packages**. Be warned that because of the simple nature of the application under test the test program is not particularly challenging. In essence the test starts to run and then immediately confirms that the application has passed the first and only test. Later we shall see how to run more interesting tests. To run the test program open AppTest.java in the edit pane and then right click to get a context menu. Choose the menu item below:

Test File Ctrl+F6

This selection causes the test program to start running and to generate a sequence of messages in the Output pane.



```
skip non existing resourceDirectory /Users/dave/Google Drive/ICP2152/Mini Projects/Mini-Project 1- Introducing Maven/Programmes/AppTest/src/test/resources

--- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @ AppTest ---
Compiling 1 source file to /Users/dave/Google Drive/ICP2152/Mini Projects/Mini-Project 1- Introducing Maven/Programmes/AppTest/target/test-classes

--- maven-surefire-plugin:2.10:test (default-cli) @ AppTest ---
Surefire report directory: /Users/dave/Google Drive/ICP2152/Mini Projects/Mini-Project 1- Introducing Maven/Programmes/AppTest/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running AppTest
main
Hello World
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.063 sec <<< FAILURE!

Results :

Failed tests:   testMain(AppTest): The test case is a prototype.

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0

------------------------------------------------------------------------
BUILD FAILURE
------------------------------------------------------------------------
Total time: 1.347s
Finished at: Fri Jan 26 09:34:03 GMT 2018
Final Memory: 18M/314M
------------------------------------------------------------------------
Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.10:test (default-cli) on project AppTest: There are test failures.

Please refer to /Users/dave/Google Drive/ICP2152/Mini Projects/Mini-Project 1- Introducing Maven/Programmes/AppTest/target/surefire-reports for the individual test
-> [Help 1]

To see the full stack trace of the errors, re-run Maven with the -e switch.
Re-run Maven using the -X switch to enable full debug logging.

For more information about the errors and possible solutions, please read the following articles:
[Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
```

```
No test passed, 1 test failed. (0.062 s)
  AppTest  Failed
    testMain  Failed: The test case is a prototype.
```

```
main
Hello World
```

Figure 6 Outputs from AppTest.java

Notice the test statistics listed under Results. Although this may not seem terribly interesting or indeed useful later exercises will demonstrate that the test framework generated by Maven can be a real asset for any working programmer. The default behaviour is fail all the tests, you will see this under the AppTest.java, you will see how to fix this next.

One final point to note is that the test programs run within a software framework known as Junit. This framework is located in the **Test Dependencies**. Open the folder and the associated jar file to get an idea of the framework contents.

Exercise 3: Testing BankAccount (20%)

In this exercise we use the Maven tool to generate a test class for a more realistic example. In fact, the class we shall test is our old friend `BankAccount` as implemented by Cay Horstmann in *Big Java:Late Objects*. Follow the steps outlined below:

- Create a Maven project called **BankAccount**;
- Store **BankAccount.java** in the **Source Packages**;

Your project should resemble the one below:

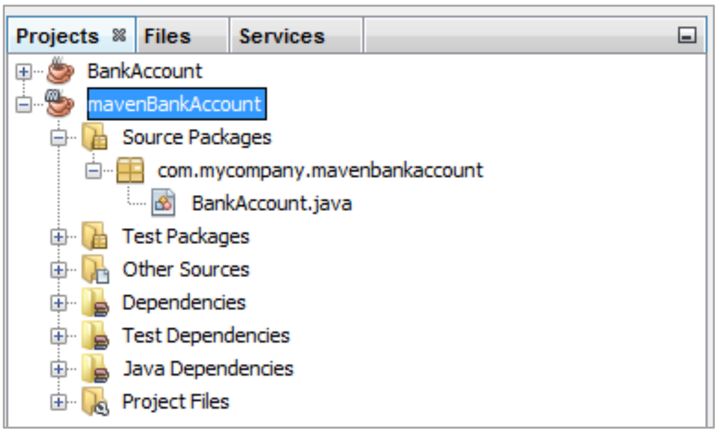


Figure 7 Project Structure of mavenBankAccount

Having placed the source code in the project we can now build the test class. To do this simply right-click on **Test Packages** and select **New → Test for Existing Class**

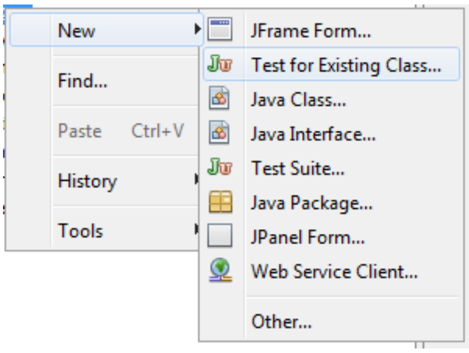
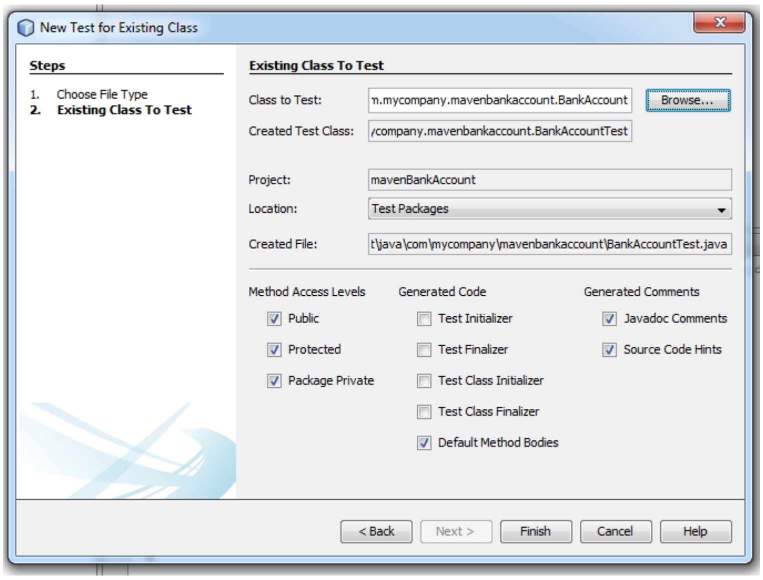


Figure 8 Menu Associated with Test Packages

This will show a new dialog to setup the test, is SHOULD automatically setup the Junit test, but might ask you to specify the details (see Fig 9)which requires you to supply the name of the class to be tested. Use the **Browse…** button to inspect **Source Packages** and then select **BankAccount.java**. Notice how this results in a fully qualified class name. Having settled on the class name click finish. Maven will respond by scanning the project and generating a test file called **BankAccountTest.java**.

The dialog box to create a test class is shown below. When you create this class do not for the moment alter any of the default settings.



New Test for Existing Class

Steps

1. Choose File Type
2. **Existing Class To Test**

Existing Class To Test

Class to Test: n.mycompany.mavenbankaccount.BankAccount

Created Test Class: /company.mavenbankaccount.BankAccountTest

Project: mavenBankAccount

Location: Test Packages

Created File: t\java\com\mycompany\mavenbankaccount\BankAccountTest.java

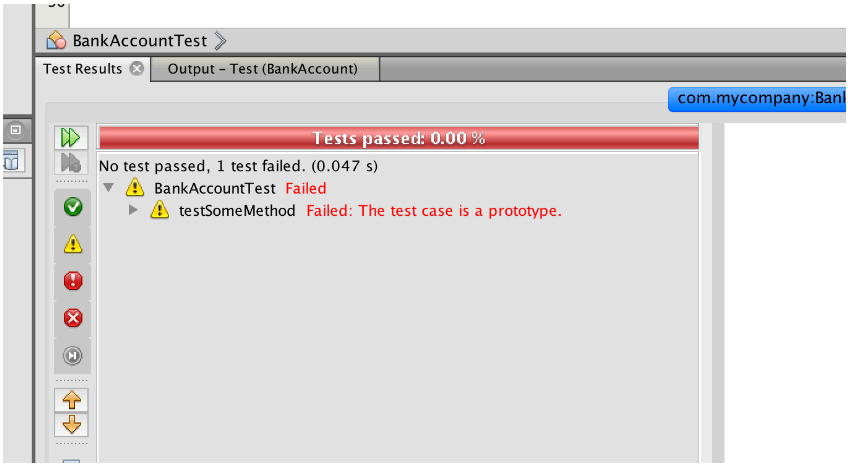
Method Access Levels
- ☑ Public
- ☑ Protected
- ☑ Package Private

Generated Code
- □ Test Initializer
- □ Test Finalizer
- □ Test Class Initializer
- □ Test Class Finalizer
- ☑ Default Method Bodies

Generated Comments
- ☑ Javadoc Comments
- ☑ Source Code Hints

Figure 9 Setting Up a Test Class

Even though this test class contains default code and so is not particularly useful it will nevertheless actually execute and generate some test results. To run open the test class in the editor pane and right-click, then select the menu option **Test**. You should get the following response:



BankAccountTest

Test Results | Output – Test (BankAccount)

com.mycompany:Banl

Tests passed: 0.00 %

```
No test passed, 1 test failed. (0.047 s)
  BankAccountTest Failed
    testSomeMethod Failed: The test case is a prototype.
```

Figure 10 Test Results

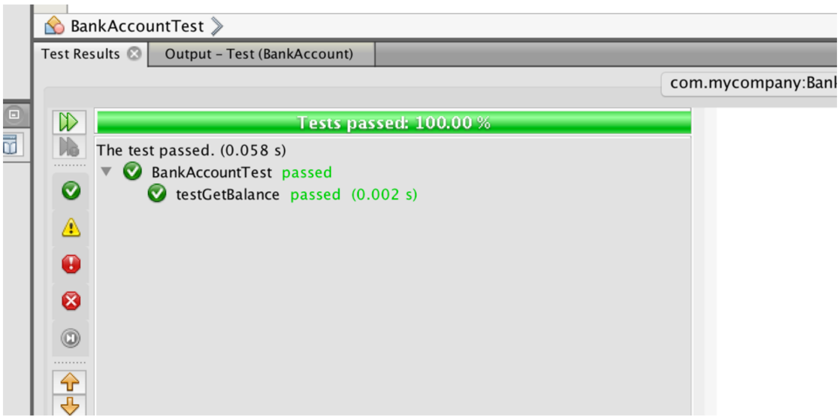
As you will see our class has failed every test. To understand why look at the test code. The test class consists of a set of methods, one for each of the methods in the class **`BankAccount`**. Each test method concludes with a call to the method **`fail()`**; this method is defined in the package **`org.junit.framework`** and if called will generate a test failure message as illustrated above. Whilst this complies the maxim that “In the beginning the class should fail every test” it will be of more use if we can develop tests that actually test the specific logic of the various **`BankAccount`** methods. To do this we can no longer rely on Maven but will have to write some code ourselves.

Exercise 4: Modifying BankAccountTest (10%)

Using the code below modify the method **`testGetBalance()`**

```
     /**
    * Test of getBalance method, of class BankAccount.
    */

 public void testGetBalance() {
     System.out.println("getBalance");
     BankAccount instance = new BankAccount(500);
     double expResult = 500.0;
     double result = instance.getBalance();
     assertEquals(expResult, result, 0.0);
 }
```

Run the test program again and you should get the following Test Results:



Tests passed: 100.00 %

The test passed. (0.058 s)

BankAccountTest passed

testGetBalance passed (0.002 s)

Figure 11 Modified BankAccountTest

Notice that the test program now reports that the program under examination, namely the class **`BankAccount`**, has passed one test. Using this as an example modify the other test methods so that meaningful tests are performed. To begin with use only the method **`assertEquals`** from the class **`junit.framework.Assert`**.

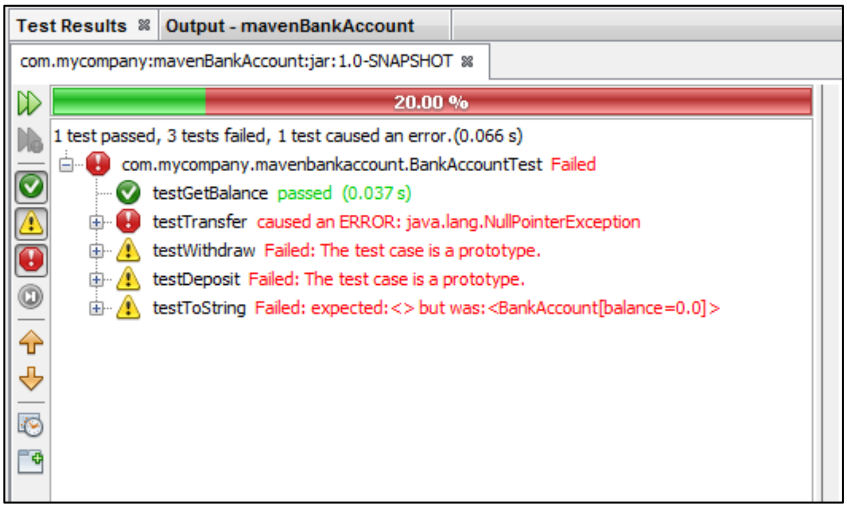
Finally, investigate the methods assertTrue and assertFalse. Can you use these methods to provide some additional tests?

Exercise 5: Modifying BankAccount (10%)

In Big Java (4th edition), Horstmann provides an additional method for BankAccount called transfer. The code for this method is listed below:

```
/**
* Transfers from this to other
* @param amount the sum to be transferred
* @param other the account into which money is transferred
*/
public void transfer(double amount, BankAccount other)
{
   withdraw(amount);
   other.deposit(amount);
}
```

Add this method to the class BankAccount and revise the test program so that appropriate tests are performed on the updated version of this class. Notice that if you simply re-run the test program you do not even get a default test method for transfer. Figure out what needs to be done and then do it!


Test Results
Output - mavenBankAccount
com.mycompany:mavenBankAccount:jar:1.0-SNAPSHOT
20.00 %

```
1 test passed, 3 tests failed, 1 test caused an error.(0.066 s)
com.mycompany.mavenbankaccount.BankAccountTest Failed
   testGetBalance passed (0.037 s)
   testTransfer caused an ERROR: java.lang.NullPointerException
   testWithdraw Failed: The test case is a prototype.
   testDeposit Failed: The test case is a prototype.
   testToString Failed: expected:<> but was:<BankAccount[balance=0.0]>
```

Figure 12 One Test Passed

Exercise 6: Providing a GUI for BankAccount (10%)

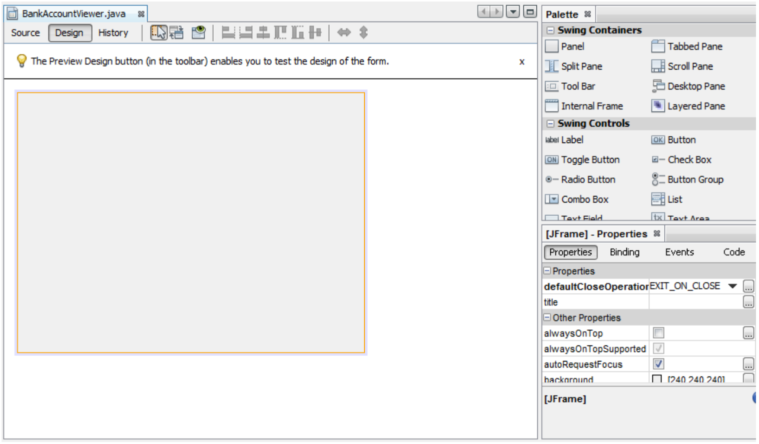
In this exercise we develop a simple graphical user interface to enable the user to perform operations on a bank account. In particular, the interface will support the following operations:

- deposit a specified amount of money into the account;
- withdraw a specified amount of money from the account;
- obtain and disBplay the balance in the account.

To create the interface we make use of the builder tool provided by NetBeans. To start using the tool follow the instructions below

- Right click on the package containing the source file BankAccount.java;
- Select **New → JFrameForm**

You will then be prompted for the name of the class which will generate the **`JFrame`**. Following Horstmann, I used the name **`BankAccountViewer`**. If everything has been done correctly the frame class should be opened in the design view which consists of an empty frame and a palette of widgets on the right hand side of the pane. This is illustrated below:



The Preview Design button (in the toolbar) enables you to test the design of the form.

defaultCloseOperation EXIT_ON_CLOSE
title
alwaysOnTop ☐
alwaysOnTopSupported ☑
autoRequestFocus ☑
background [240,240,240]

Figure 13 Design View of BankAccountViewer

Using the facility of the builder tool create a frame similar to the one illustrated below.

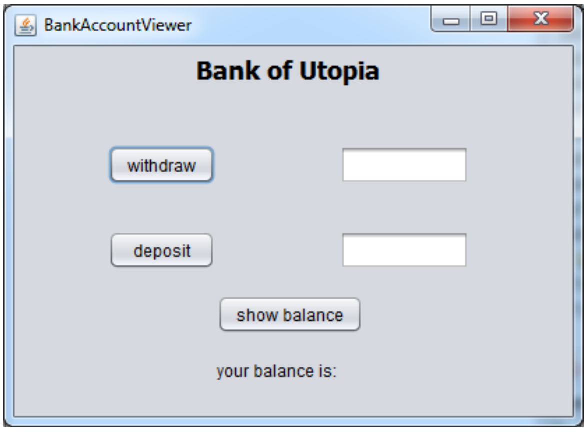


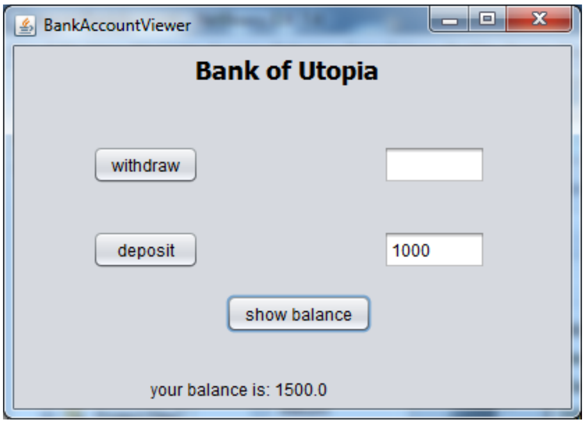
Figure 14 Builder Generated GUI

Once you have fine tuned the layout of the interface components you should provide the necessary functionality.

This will be done by

- Declaring and creating a **`BankAccount`** object as an instance variable for **`BankAccountViewer`**;
- Modifying the code for **`BankAccountViewer`** to provide appropriate handlers.

The operation of the GUI is illustrated below:



Bank of Utopia

1000

your balance is: 1500.0

Figure 15 Make a Deposit and Check Balance

In the above example the account was created with £500 and then a deposit of £1,000 was made so that the new balance is now £1,500 as displayed.

Exercise 7: Creating a Dependency (10%)

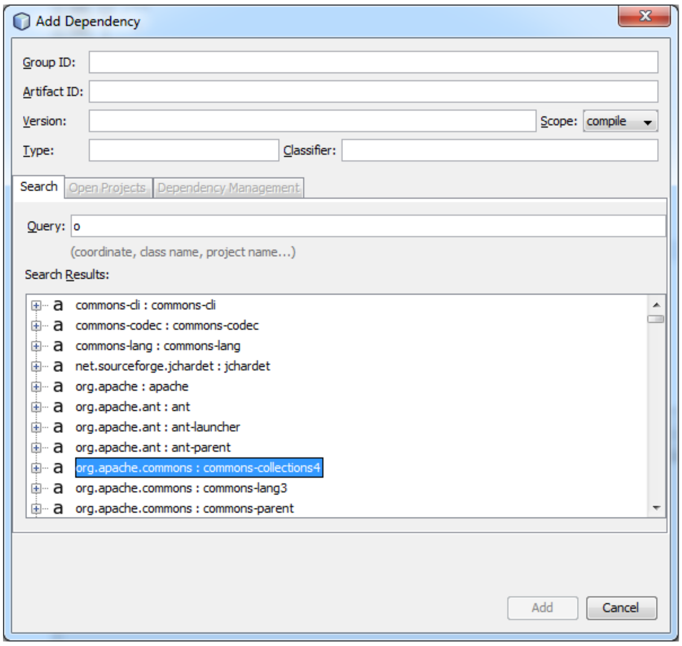
In this exercise we shall add a dependency, namely a .JAR file supplied by Apache Commons, to our Maven project. For those of you who don't know, Commons is the name of a project run by the Apache Software Foundation, the purposes of which is to supply reusable, open source Java software. The particular component we are interested in is the API known as **`Collections 4.0`**. For information about this component visit the site below:

http://commons.apache.org/proper/commons-collections/javadocs/api-release/index.htmle

In Maven the management of dependencies is relatively straightforward. Follow the steps below:

- Right-click on the dependencies folder;
- Select **Add Dependency…**
- Type **`org.apache.commons`** in the Query box
- Select **`org.apache.commons: commons-collections4`**
- Expand this item and select the jar file

The **Add Dependency** dialog is shown below.



Scope: compile

Query: o

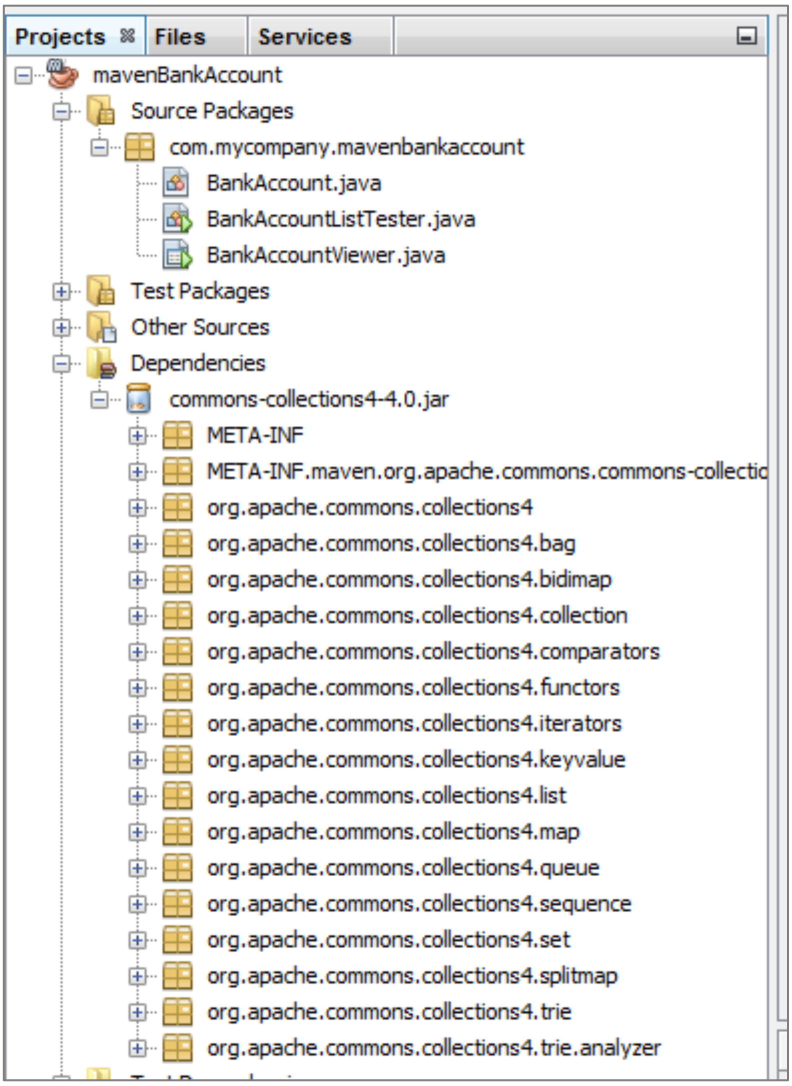
(coordinate, class name, project name…)

commons-cli : commons-cli
commons-codec : commons-codec
commons-lang : commons-lang
net.sourceforge.jchardet : jchardet
org.apache : apache
org.apache.ant : ant
org.apache.ant : ant-launcher
org.apache.ant : ant-parent
org.apache.commons : commons-collections4
org.apache.commons : commons-lang3
org.apache.commons : commons-parent

Figure 16 Selecting a Commons Component

The dependency will now be automatically created in the project. The result of this process is illustrated on the next page. Notice that the .JAR file has been expanded so that the individual packages which it contains are displayed.

The names of the packages should be familiar to you because they are intended as extensions to the Java Collections Framework.



```
mavenBankAccount
    Source Packages
        com.mycompany.mavenbankaccount
            BankAccount.java
            BankAccountListTester.java
            BankAccountViewer.java
    Test Packages
    Other Sources
    Dependencies
        commons-collections4-4.0.jar
            META-INF
            META-INF.maven.org.apache.commons.commons-collectio
            org.apache.commons.collections4
            org.apache.commons.collections4.bag
            org.apache.commons.collections4.bidimap
            org.apache.commons.collections4.collection
            org.apache.commons.collections4.comparators
            org.apache.commons.collections4.functors
            org.apache.commons.collections4.iterators
            org.apache.commons.collections4.keyvalue
            org.apache.commons.collections4.list
            org.apache.commons.collections4.map
            org.apache.commons.collections4.queue
            org.apache.commons.collections4.sequence
            org.apache.commons.collections4.set
            org.apache.commons.collections4.splitmap
            org.apache.commons.collections4.trie
            org.apache.commons.collections4.trie.analyzer
```

Figure 17 Listing Packages from the JAR

The package we are interested in is org.apache.commons.collections4.list because it contains a number of classes that we can use to develop our bank account project. To begin with we shall work with the class **`GrowthList`** stored in the package:

`org.apache.commons.collections4.list`

To test that classes in the dependency can be accessed you should create a Java **`main`** class in **Source Packages**. Then, inside the **`main`** method add code to

- Create an instance of **`GrowthList`** called **`bank`** ;
- Create three instances of **`BankAccount`** and store them in **`bank`**;
- Display the contents of **`bank`**;

Note that the contents need only be displayed in the NetBeans Output pane.

Exercise 8: Using a Lightweight Database (10%)

HSQLDB is a relational lightweight database engine written in Java, with a JDBC driver, conforming to ANSI SQL:2008. It offers a small, fast multithreaded and transactional database engine with in-memory and disk-based tables and also supports embedded and server modes. In addition the database provides a powerful command line SQL tool and a set of simple GUI query tools. For more information about HSQLDB visit the official site

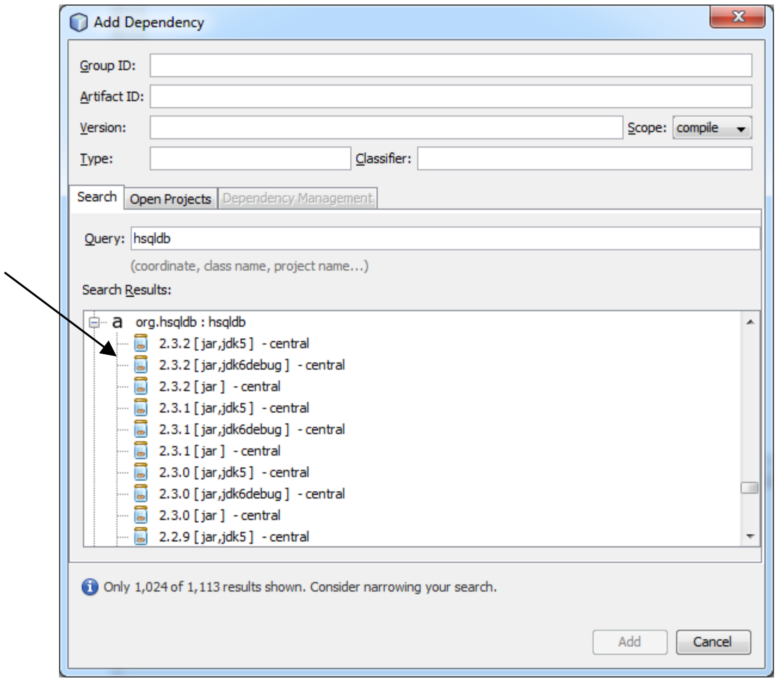
http://hsqldb.org/

Whilst it is possible to install HSQLDB by downloading the distribution and manually setting the class path it is easier to let Maven to do this work for us.

Follow the steps below:

- Right-click on the dependencies folder;
- Select **Add Dependency…**
- Type **`org.hsqldb`** in the Query box
- Select **`hsqldb-2.3.2-jdl5.jar`**
- Click the **Add** button

This selection is illustrated in the figure below:



Scope: compile

Query: hsqldb

(coordinate, class name, project name…)

Search Results:

```
org.hsqldb : hsqldb
    2.3.2 [ jar,jdk5 ]  - central
    2.3.2 [ jar,jdk6debug ]  - central
    2.3.2 [ jar ]  - central
    2.3.1 [ jar,jdk5 ]  - central
    2.3.1 [ jar,jdk6debug ]  - central
    2.3.1 [ jar ]  - central
    2.3.0 [ jar,jdk5 ]  - central
    2.3.0 [ jar,jdk6debug ]  - central
    2.3.0 [ jar ]  - central
    2.2.9 [ jar,jdk5 ]  - central
```

Only 1,024 of 1,113 results shown. Consider narrowing your search.

Figure 18 Using Query to Locate HSQLDB

If all has gone according to plan a new dependency called **`hsqldb-2.3.2-jdk5.jar`** should have been added to your project. In Maven terminology the **`.jar`** file is said to have been placed in your local repository (on my machine this is located in the directory M:\Windows_Data\.m2).

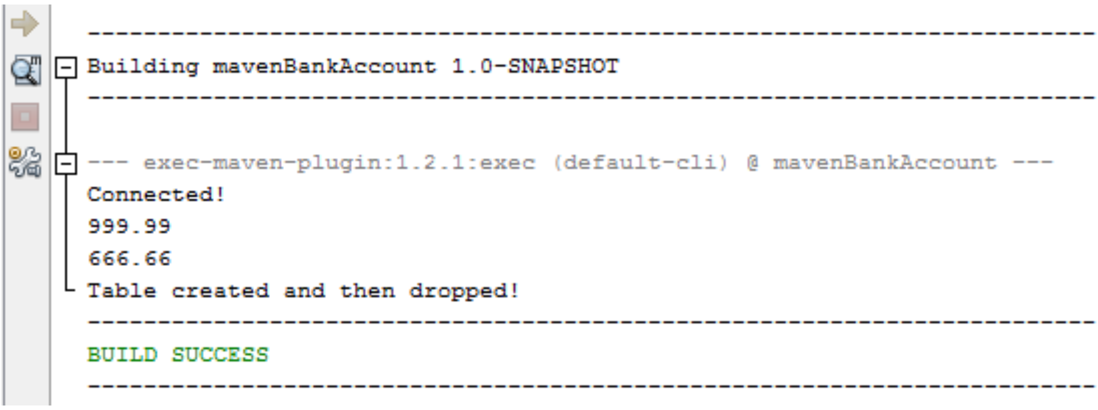
Exercise 9: Creating a Toy Database (10%)

This exercise requires you to use HSQLDB to build a toy database. To do this you will need to get the program **HSQLTester** running (see Appendix 2). This test program requires the following items:

- **SimpleDataSource.java** (see Horstmann's chapter on Database Programming)
- **database.properties**

For the purposes of this exercise store the **database.properties** file in the same folder as your other Java programs. Although this approach is not recommended it has the merit of simplicity. However, not all simple solutions are good solutions and in the next exercise we shall adopt a more sophisticated approach to the management of resources such as properties files.

If the test has run successfully you should get the following output. Notice that the two account balances are displayed.



```
------------------------------------------------------------------------
Building mavenBankAccount 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ mavenBankAccount ---
Connected!
999.99
666.66
Table created and then dropped!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Figure 19 Testing HSQLDB

Having got this much working we can be fairly sure, in an unscientific sort of way, that HSQLDB has been installed correctly.

Exercise 10: Working With POM Files (10%)

The previous exercise introduced the use of a property file to contain essential configuartion information. In this exercise we use the POM file to perform essentially the same job, although as we shall see the use of this Maven element brings with it some additional advatages. To complete this exercise work through the following steps.

Step 1: Modify Test Program

Insert the code fragment below into **HSQLTester**

```
...
InputStream =
   SQLTester.class.getResourceAsStream("/database.properties");
SimpleDataSource.init(stream);
Connection conn = SimpleDataSource.getConnection();
```

This fragment ensures that data is now read from the properties file using an **`InputStream`** rather than a **`FileInputStream`**. Notice that you will also have to modify the method **`SimpleDataSource.init()`** so that it takes an **`InputStream`** as parameter and not a file name.

Step 2 Edit the POM File to include a <build> section

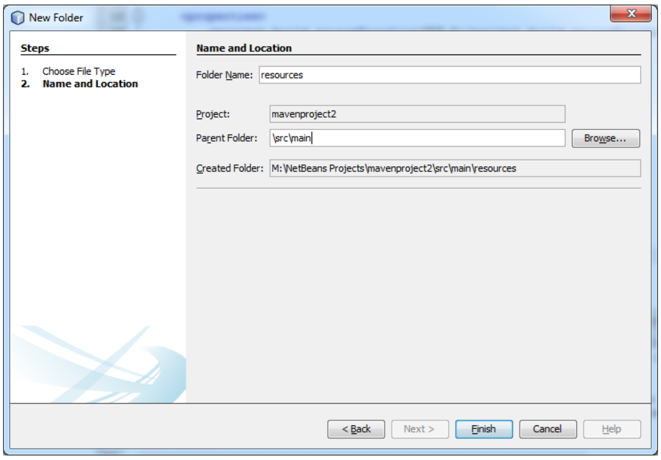
Place the following XML snippet in the POM file for this project.

```
<build>
   <resources>
      <resource>
         <directory>src/main/resources</directory>
         <filtering>true</filtering>
       </resource>
   </resources>
</build>
```

The fragment should be placed just below the end of the **`<properties>`** node. Now build the project again and note that Maven complains about a missing directory. To resolve this problem go to Step 3.

Step 3 Create the resources folder

To do this right click the project icon and complete the dialog below as illustrated:



Steps

1. Choose File Type
2. **Name and Location**

Name and Location

Folder Name: resources

Project: mavenproject2

Parent Folder: \src\main

Created Folder: M:\NetBeans Projects\mavenproject2\src\main\resources

Figure 20 Create resources folder

Once this is done the project view will be automatically modified and a new folder called **Other Sources** will be created to contain **resources**.

Step 4 Edit the POM file to include a <profiles> section

Here we provide Maven with a profile which will ensure that during the build process an appropriate properties file is generated. Open the POM.xml file and copy the fragment below into this file.

```
<profiles>
   <profile>
      <id>hsql</id>
      <dependencies>
         <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>2.3.2</version>
            <classifier>jdk5</classifier>
         </dependency>
      </dependencies>
      <properties>
         <jdbc.url>jdbc:hsqldb:mem:testDB</jdbc.url>
         <jdbc.username>SA</jdbc.username>
         <jdbc.password></jdbc.password>
         <jdbc.driver>org.hsqldb.jdbc.JDBCDriver</jdbc.driver>
      </properties>
    </profile>
 </profiles>
```

Also a create a text file called **database.properties** containing the four lines below and place this in the folder **resources** which you have just created. To do this right click on the resources folder just created and select **New → Properties File** .

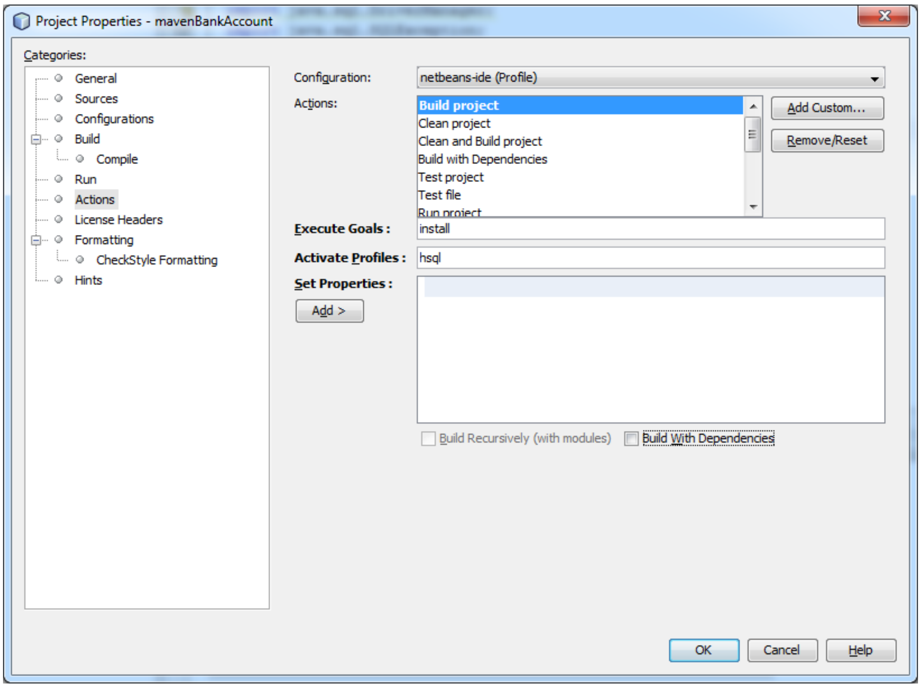
```
jdbc.url=${jdbc.url}
jdbc.username=${jdbc.username}
jdbc.password=${jdbc.password}
jdbc.driver=${jdbc.driver}
```

Having done all this you should now delete database.properties from the Source Packages folder (remember in Exercise 9 we created and stored this file alongside the other Java source files). We no longer need this file because it will be generated by Maven during the build process on the basis of information supplied in the **<profile>** section of the POM file.

Step 5 Use Project Properties dialog to Set Up Profiles

Finally we need to tell Maven to use the hsql profile in the course of various critical actions. The instructions below will set one of the actions, namely Build, you will need to deal with the others.

Right click on the project and select **Properties**. Then click on **Build** project and enter hsql in the **Activate Profiles** box. Click on **OK**. This profile has now been activated for the **Build** project option. You should repeat the process for the following Actions: Clean and Build project, Build with Dependencies.



Project Properties - mavenBankAccount

Configuration: netbeans-ide (Profile)

Actions: Build project

Execute Goals : install

Activate Profiles : hsql

Set Properties :

□ Build Recursively (with modules) □ Build With Dependencies

Figure 21 Activating hsql Profile

You should now be in a position to run the test program and hopefully obtain the same results as in Exercise 9.

Having been guided through this process you are now required to add an additional profile to your POM file so that a programmer can switch between using hsql and mysql by simpling activating different profiles.

Appendix 1: Introduction to HSQLDB

HSQL – the "H" stands for "Hyper" - is a DBMS written purely in Java and is an example of a *lightweight* or *embedded* database. Note that HSQL can be operated in *one* of *three* modes:

- **Memory** – The database is held in memory such that when the application is closed the data is lost. This is generally only useful for applications – e.g. applets - which do not need to maintain persistent data.
- **Standalone** – HSQL runs the database engine *as part of your application program in the same Java Virtual Machine*. For most applications this mode is faster, as the data is not converted and sent over the network. The main drawback is that it is not possible by default to connect to the database from outside your application. The HSQL site suggests that in the development stage programmers should use the Server mode(see below) as this allows applications (e.g. Database Manager to inspect the contents of the database as soon as it has been created) and when development is concluded revert to Standalone.
- **Server** – Server modes provide the maximum accessibility. The database engine runs in a JVM and listens for connections from programs on the same computer or other computers on the network. Several different programs can connect to the server and retrieve or update information. Applications programs (clients) connect to the server using the HSQLDB JDBC driver. In most server modes, the server can serve up to 10 databases that are specified at the time of running the server.

In these laboratory notes the focus is on using HSQL in the more efficient *standalone mode* rather than the more accessible server mode. In consequence it will not be possible to independently inspect the contents of the databases we have created, not at least until we have a Java application up and running. However, this will not be a problem as the examples used are simple and should pose no real problems for an averagely competent programmer.

One final point before moving on. According to the HSQL manual there a number of different *types of table* available to the user of this DBMS. The following edited extract from the manual summarises the situation:

> MEMORY tables are the default type when the CREATE TABLE command is used. Their data is held entirely in memory but any change to their structure or contents *is written to the <dbname>.script file*. The script file is read the next time the database is opened, and the MEMORY tables are recreated with all their contents.
>
> CACHED tables are created with the CREATE CACHED TABLE command. Only part of their data or indexes is held in memory, allowing large tables that would otherwise take up to several hundred megabytes of memory. Another advantage of cached tables is that the database engine takes less time to start up when a cached table is used for large amounts of data. The disadvantage of cached tables is a reduction in speed. Do not use cached tables if your data set is relatively small. In an application with some small tables and some large ones, it is better to use the default, MEMORY mode for the small tables.

In this laboratory session we shall rely on the *default table type* (i.e. memory) because the table we are working on is very small. Note that the data stored in a memory table is persistent in the sense that it is generated at run-time by a script file.

What is an Embedded Database?

When a Java application accesses a HSQL database using the *embedded* JDBC driver, the HSQL engine does not run in a separate process, and there are no separate database processes to start up and shut down. Instead, HSQL runs *inside* the same Java Virtual Machine (JVM) as the application. So, HSQL becomes part of the application just like any other jar file that a Java applicationmight use.

Loading the JDBC Driver

Every driver implementation, including the HSQLDB driver, must provide a class which implements the `java.sql.Driver` interface. Another class named `DriverManager` is tasked with utilising the appropriate driver in order to connect to DBMS we specify, this class 'will try to load as many drivers as it can find and then for any given connection request, it will ask each driver in turn to try to connect to the target URL'.

We first of all need to load the implementation of the **`Driver`** interface supplied by the downloaded driver (**JAR** file). Note that 'when a Driver class is loaded, it should create an instance of itself and *register* itself with the DriverManager'. Therefore, by loading a **`Driver`** class, it should register with the `DriverManager` which in turn will utilise this to connect to the DBMS when requested. We load the driver and register it as follows:

```
Class.forName("org.hsqldb.jdbcDriver");
```

In the above, **`org.hsqldb.jdbcDriver`** is the name of the class implementing the **`java.sql.Driver`** interface. JDBC 4.0 contains a feature known as *driver auto-loading* which means that we are able to omit the above line of code, however we include this information for the purposes of backward compatibility.

Note that this is useful knowledge for when you come across a DBMS which provides a pre-JDBC 4.0 driver – something which is not uncommon. Having said that auto-loading requires someone to register the driver by setting a value in the properties file associated with the database – either as a command line argument or using the **`System.setProperty()`** method. Furthermore there are occasions where auto-loading cannot be used e.g. if the application executes inside a servlet container. So you might as well learn how to use **`Class.forName()`** !!!

Establishing A Connection

A connection to the database is represented using an instance of **`java.sql.Connection`**. The instance is obtained by invoking the **`getConnection()`** method of the **`DriverManager`** class. In fact, multiple implementations of this method are provided by the **`DriverManager`** class as the method is overloaded allowing connection information to be supplied in a variety of methods. We'lll look at the version which accepts three String objects as parameters, these being:

- The JDBC URL of the database to connect to.
- The username to authenticate to the database.
- The password to authenticate to the database.

A JDBC URL is much like any other URL you will have encountered previously, being used in order to connect to a particular resource, in this case, a database. Unfortunately, JDBC URLs are non-standardised which means that they are of a different format for every type of DBMS we wish to use, the only thing they have in common being that they must begin with the prefix 'jdbc:'.

A HSQLDB JDBC URL will take one of the following forms

`jdbc:hsqldb:file:`*`databaseName`*	Stand-alone
`jdbc:hsqldb:hsql:`*`databaseName`*	Server
`jdbc:hsqldb:mem:`*`datbaseName`*	Memory

For purposes of these assessed exercises use the following

- **`jdbc:hsqldb:file:`*`databaseName`***

Where ***databaseName*** is the name of the file in which HSQLDB will store its data (i.e. it is the name of the database). Note that you may specify the full path to the file or simply a file name on its own in which case the database will be created in the folder associated with the current NetBeans project.

To connect to the database you will need to supply a username and password. The default values for HSQLDB are listed below:

- default username is "**`sa`**"
- default password is the empty string i.e. ""

You should note that the method **`getConnection()`** may throw a **`SQLException`** in the event of error, therefore we have to declare that our constructor throws an instance of **`SQLException`**. This exception is a fairly generic type of exception and is thrown when any error relating to the use of the database occurs – this might include attempting to execute a badly constructed SQL statement or failing to connect to the database amongst other things.

Appendix 2: Test Program for HSQLDB

```
package com.mycompany.mavenbankaccount;

import java.sql.*;
import java.io.*;
import java.sql.ResultSet;

public class HSQLTester
{
    public static void main(String[] args) throws IOException,
    ClassNotFoundException, SQLException
    {

      // This is an amended version of Horstmann’s test program.

      SimpleDataSource.init("database.properties");
      Connection conn = SimpleDataSource.getConnection();

      Statement st = conn.createStatement();

      try {
          st.execute(
                  "CREATE TABLE IF NOT EXISTS
                      accounts (balance DECIMAL(5,2))");

          st.execute("INSERT INTO accounts VALUES (999.99)");
          st.execute("INSERT INTO accounts VALUES (666.66)");

          ResultSet rs  = st.executeQuery("SELECT * FROM accounts");

          while(rs.next())
          {
              System.out.println(rs.getString("balance"));
          }

          st.execute("DROP TABLE accounts");
      }
      finally
      {
          System.out.println("Table created and then dropped!");
          st.close();
          conn.close();
      }
    }
}
```