Practical 2: Testing Classes with JUnit

1 Objectives

- Review a class that will record your yearly module marks.
- See how to create a JUnit test to test the class.

2 Preparation

2.1 Setting up git.

If you are at a machine that you have not used before, you may not have a local copy of the last practical you did. So, to get back your work and add work from today's practical session, you would need to **clone** your repository.

If you have a local copy of your repository already, then please ignore the commands for cloning a repository, and move on to the section 2.1.2.

2.1.1 Cloning your repository.

You should use the following commands to clone your repository (please note that **#** precedes a comment).

- cd directory_of_your_choice
- git clone https://user_name@bitbucket.org/user_name/soft25x.git # use your own repository address for this
- ls # this should show a list of directories in *directory_of_your_choice*, soft25x should be here if cloning worked.

2.1.2 Creating directory for today's practical session.

This assumes you have your repository in your local machine. Now, you would create a new directory for today's session (please note that **#** precedes a comment).

- cd soft25x # make sure you use your repository directory.
- mkdir Week02 # create a directory called *Week02*.
- ls # check that the directory was created.

Note: It is sometimes useful to use commands from the PowerShell. A list of common commands is available on the digital learning environment (DLE) module page.

2.1.3 Unzip provided code, and update the repository.

You will use the ***soft25x\Week02*** directory for today's session. Now, download the **JunitPracticalData.zip** from digital learning environment (DLE) module page, and follow the steps below.

- Unzip the **JUnitPracticalData.zip** file to: ***soft25x\Week02***.

Now, issue the following commands on the PowerShell window.

- cd soft25x # only issue this if you are not already in that directory
- git add Week02
- git commit –m "added the provide code to repository"
- git push

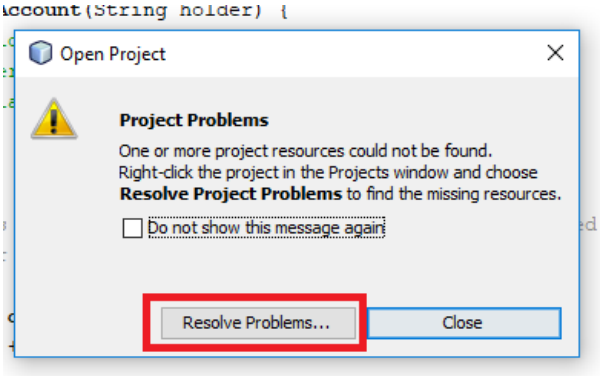
Now, you should verify on bitbucket website that your repository now contains the code you just uploaded.

2.1.4 Continuing with the provided code.

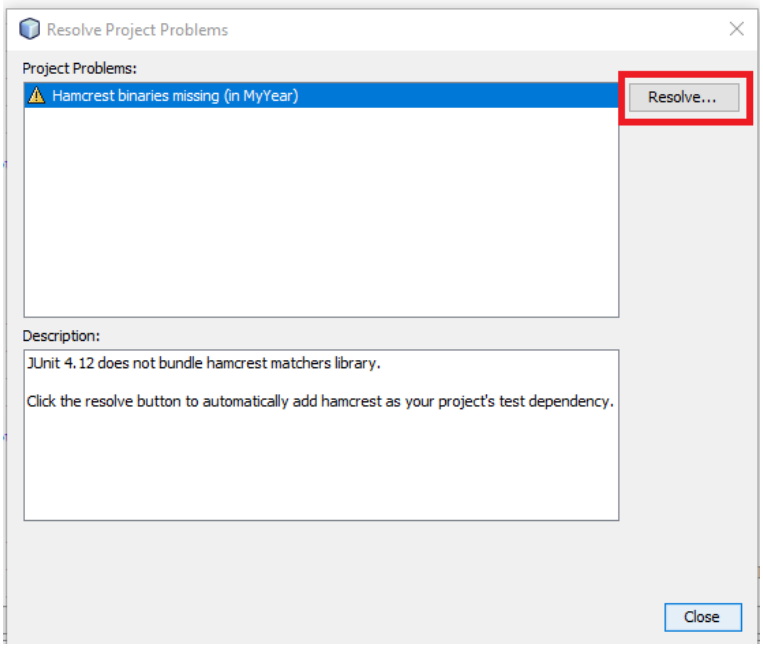
Now, follow the instructions below.

- **Start** menu, **University Software > N > NetBeans**
- Open the **MyYear** project.

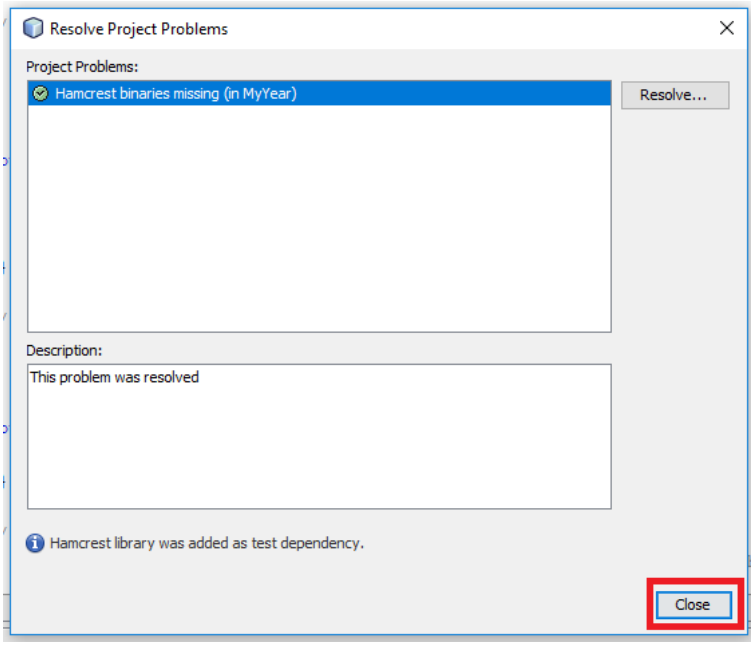
Trying to open the project may result in the following warning message, asking you to resolve project problems. Please press "Resolve Problems…" button, highlighted with a red box below.



It will now show you the exact problems: *Hamcrest binaries missing (in MyYear)*. Please press the "*Resolve*" button.



The issues should now be resolved. Please close the window, and carry on with the rest of the exercise.



The project contains a class with the constructor and accessor methods that were created for you. You will now add a couple more methods (addModuleMark and calculateAverageSoFar as shown in the class diagram below) and then set up the JUnit code.

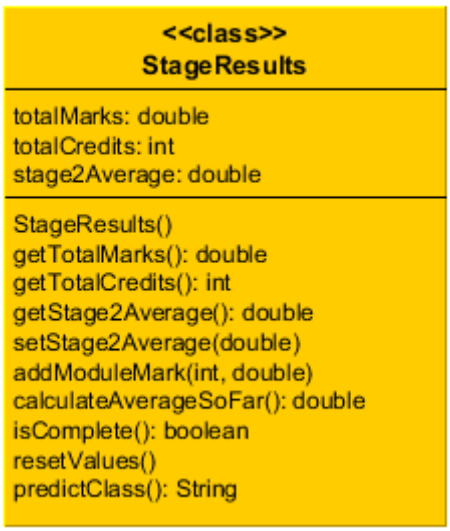


Figure 1 A class diagram for the StageResults class. In the properties section, the types of variables are given after the colon (:). In the methods section, the types of input variables are given in brackets, and the types of returned variables are given after the colon (:).

The stage requires you to have studied a total of 120 credits. Modules may be 10, 20 or 40 credits in total. As you get your marks, you can enter the overall mark for the module and also the number of credits for that module.

The class can calculate your average mark so far for the number of credits you have achieved, and once all the marks have been entered (i.e. the total number of credits added is 120) it can predict your final classification. Normally, the final degree classification will be calculated from 30% of your stage 2 marks plus 70% of your final stage marks, so if you enter your stage 2 mark it will be able to use that in the prediction.

When you add a module, its marks need to be added into the total so far, but with different modules being worth different number of credits, the class needs to be able to make an allowance for this. So a 10-credit module is just added as it is, but a 20-credit module is equivalent to two 10 credit modules, so it has to add that module's mark in twice. For a 40-credit module it needs to add it in four times.

At the end of this tutorial, you will be able to view an example of a GUI application that uses the class. It also saves the data to a file so that it can be used to accumulate the data over a period of time.

3 Completing the StageResults class

Open the class and look through the code to see the methods that are already provided. The next step is to implement the remaining methods.

3.1 addModuleMark(int, double)

This method will add in a module mark, but also allow for the fact that 20 credits counts double, and 40 credits counts four times. To achieve this we multiply the mark by (number of credits ÷ 10), so for 10 credits it is x 10÷10, for 20 credits it is x 20÷10, for 40 credits it is 40÷10.

This is the code to enter:

```
public void addModuleMark(int credits, double mark) {
    totalCredits += credits;
    totalMarks += mark * (credits / 10);
}
```

Reminder: Did you update your git repository?

3.2 calculateAverageSoFar()

This method calculates the average mark so far. This is found by dividing the total marks by the number of modules added so far. So if the total credits is currently 60 that is equivalent to 6 modules.

Add this code:

```
public double calculateAverageSoFar() {
    double average;

    average = totalMarks / (totalCredits / 10.0);
    average = Math.round(average * 100) / 100.0;

    return average;
}
```

Reminder: Did you update your git repository?

3.3 predictClass()

This method predicts the classification based on the average so far, if all marks have been entered. If the number of credits is less than the total required, the classification will not be calculated.

First, the average for stage 3 is calculated, and then if the stage 2 average has been entered it calculates the final overall mark as 70% of stage 3 + 30% of stage 2. The classification is then determined based on the range in which the overall average falls.

```
public String predictClass() {
    double overallAverage = calculateAverageSoFar();
    String degree;
```

```
    if (stage2Average != 0)
        overallAverage = Math.round(overallAverage * 0.7 * 100) / 100.0
                + Math.round(stage2Average * 0.3 * 100) / 100.0;

    if (totalCredits < MAXCREDITS)
        degree = "Insufficient credits";
    else if (overallAverage == 0)
        degree = "No marks!";
    else if (overallAverage < 40)
        degree = "FAIL";
    else if (overallAverage < 50)
        degree = "3rd";
    else if (overallAverage < 60)
        degree = "Lower 2nd";
    else if (overallAverage < 70)
        degree = "Upper 2nd";
    else
        degree = "1st";

    return degree;
}
```

- Reminder: Did you update your git repository?

3.4 Next Step

Ensure all files are saved before proceeding to the next section where we will develop the test class.

4 Developing a JUnit test

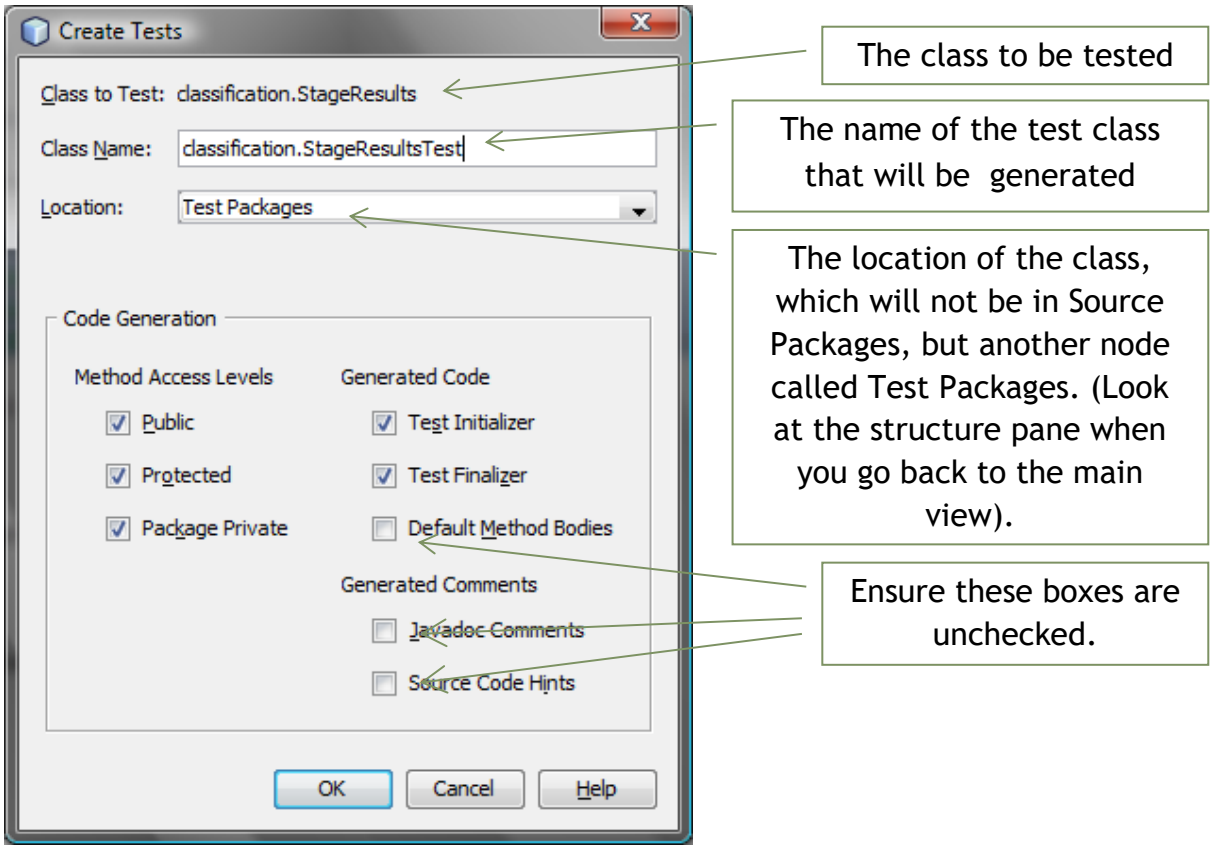
One of the problems with using a console application is that you have to check that expected and actual results match – laborious and easy to see what you want to see!

JUnit will do the checking for you and give a simple report showing whether each test has passed or failed. We will get NetBeans to create the framework for our tests; we then add the details within the test methods. The set and get methods and the constructor were generated by NetBeans so we won't test them.

4.1 Preparing the tests

- In the structure pane, right-click **StageResults.java**
- Select **Tools** ▶
- Select **Create JUnit Tests (could also be named: Create/Update Tests)**
- A dialog may ask which version; ensure the latest one is checked and click **Select**

- This dialog is shown:



- Click **OK**
- Expand the **Test Packages nodes**, then expand **classification**. This shows the location of the test class.

The class is shown in the code pane, and currently consists of stubs for the various methods. The *@Before* and *@After* methods are those that will be run before and after the test methods. The *@Test* methods are designated as tests.

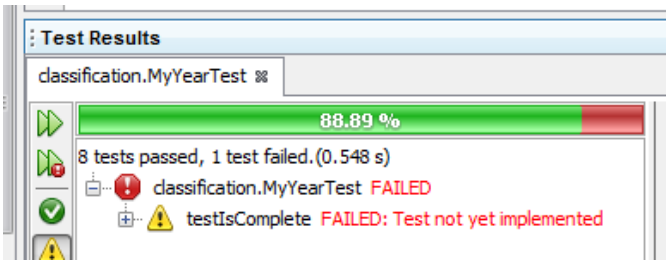
Before we start entering code, we will put a FAIL message in each of the tests that we will be implemented, to show that they are yet to be coded.

- In testIsComplete() (*i.e. the test method for our isComplete() method*), add this line of code:

```
@Test
public void testIsComplete() {
    fail("Test not yet implemented");
}
```

- Save all files.
- Run the test by right-clicking **StageResults** and selecting **Test File**
- A test results pane will appear showing these results:

The methods that don't contain a fail message are assumed to have passed. The one that has the fail message shows the result and shows the message that was included.



Now add the same message to the following:

- testResetValues()
- testAddModuleMark()
- testCalculateAverageSoFar()
- testPredictClass()

The basic set and get methods will be ignored, and will be assumed to have passed. We will leave the stubs there in case they are needed in future.

Run the test again – there should now be five failures.

- Reminder: Did you update your git repository?

Note: There are some hints on this practical session at the end of this document.

4.2 Adding some data

In order to run tests, we need some objects that include some data for testing, as shown below.

Declaration:

```
public class StageResultsTest {
    private StageResults empty;     // will have no credits and no marks
    private StageResults full;      // will have 120 credits and marks
    private StageResults halfFull;  // will have 60 credits and some marks
```

Method setup() (*NOT setUpClass()*):

```
@Before
public void setUp() {
    // empty - object that starts with default values
    empty = new StageResults();

    // full - object with 120 credits-worth of marks but no
    // initial stage2Average
    full = new StageResults();
    full.addModuleMark(120, 50.0);

    // halfFull - object with 60 credits worth of marks and
    // no initial stage2Average
    halfFull = new StageResults();
```

```
    halfFull.addModuleMark(60, 50.0);
}
```

- Reminder: Did you update your git repository?

4.3 Testing *isComplete()*

This is a fairly simple one to start with. Looking at the data, the object should return:

empty ⇒ false
full ⇒ true
halfFull ⇒ false

So the code for the **empty** object will look like:

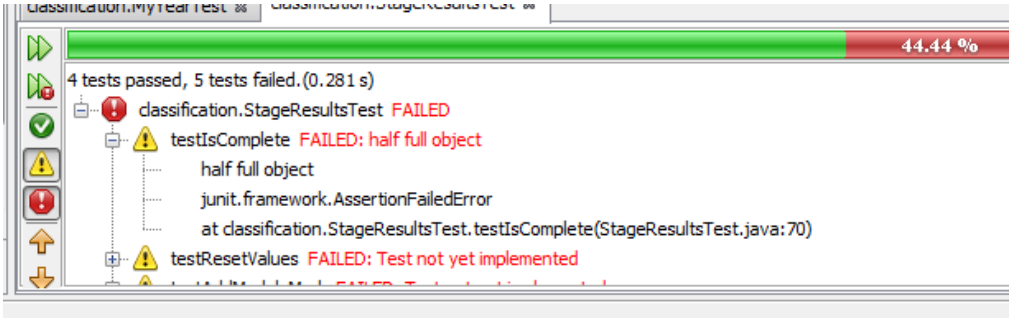
```
public void testIsComplete() {
    fail("Test not yet implemented");
    System.out.println("Testing isComplete");

    // Check that the empty object is 'not complete'
    assertFalse("empty object", empty.isComplete());
}
```

- Add the code as show as above, and using the same pattern, also add the tests for the **full** and **halfFull** objects.
- Run the test. This time there will be just 4 tests that fail.

Next, introduce a bug in the **StageResults** class - change the value of the **MAXCREDITS** constant from 120 to 60 and run the test again.

If you expand the **testIsComplete** node in the test results, this is what you should see:



Put the correct value back in the constant.

- Reminder: Did you update your git repository?

4.4 Testing *resetValues()*

This method does not return any values so we need to tackle it slightly differently. This time, we will just use one of the objects (that does currently have some non-zero data), send it the resetValues() message. We then need to use getTotalCredits()

and getTotalMarks() to check that all have been reset to zero. After the test, we will reset the object back to its original state.

Add the code:

```
@Test
public void testResetValues() {
    fail("Test not yet implemented");
    System.out.println("Testing resetValues");

    // Set the state of the 'full' object to zeroes
    full.resetValues();

    // Set expected results
    int expIntResult = 0;
    double expDoubleResult = 0.0;

    // Now check each attribute to test that the reset has worked
    assertEquals("credits", expIntResult, full.getTotalCredits());
    assertEquals("total", expDoubleResult, full.getTotalMarks(), 0.0);

    // Put the 'full' object back to its original state
    full.addModuleMark(120, 50.0);
}
```

Test the file, there will now be three failures instead of four.

- Reminder: Did you update your git repository?

4.5 Testing *addModuleMark(int, double)*

Now work out some testing for the method that adds module marks.

For this test you could use the *empty* object, which currently has no marks and no credits. When you add a module mark there is no return value, so like the last test, you will need to use the method and then getCreditsSoFar() and getTotalSoFar() to check that the marks and credits have been properly added.

Note that at this point we are not interested so much in validation (i.e. being able to add more than 120 credits-worth of marks) but whether the correct outputs are given for the inputs used. We'll be looking at the validation aspects a little later.

You should test the addition of a 10 credit module, a 20 credit module, and a 40 credit module. Write down the values to add and the expected results before starting the code. At the end of the test method, use *resetValues()* to restore the *empty* object to its original state.

- Reminder: Did you update your git repository?

4.6 Testing *calculateAverageSoFar()*

This method performs calculations and we need to use a variety of test cases in order to check that the correct results are produced in various cases, e.g. boundary cases and cases where mid-range values are used.

Some suggested test cases are given below, and have been implemented in the sample solution in case you want some hints. The code for the first three cases is shown below to get you started. Click here if you want to see the code for the rest.

TEST	COMMENTS	EXPECTED RESULT
empty object – no credits and no marks	use as-is	0.0
full object –120 credits of marks all at 50%	use as-is	50.0
full object – 120 credits of marks all at 100%	use *resetValues()* to reset to zero, then *addModuleMarks(120, 100.0)* before testing	100.0
full object – 120 credits at 43.92%	use *resetValues()* to reset to zero, then *addModuleMarks(120, 43.92)* before testing then restore original values (*using resetValues()* and *addModuleMarks(120, 50.0)*)	43.92
halfFull object – 60 credits at 50%	use as-is	50.0
halfFull object – 60 credits at 100%	use *resetValues()* to reset to zero, then *addModuleMarks(120, 100.0)* before testing	100.0
halfFull object – 60 credits at 64.77%	use *resetValues()* to reset to zero, then *addModuleMarks(60, 64.77)* before testing then restore original values (*using resetValues()* and *addModuleMarks(60, 50.0)*)	64.77

First case:

```
// Test with no credits and no marks
assertEquals("empty", 0.0, empty.calculateAverageSoFar(), 0.0);
```

Second case:

```
// Test with 120 credits all at 50%
assertEquals("full @ 50%", 50.0, full.calculateAverageSoFar(), 0.0);
```

Third case:

```
// Test with 120 credits all at 100%
full.resetValues();
full.addModuleMark(120, 100.0);
```

```
assertEquals("full @ 100%", 100.0, full.calculateAverageSoFar(), 0.0);
```

Add the above and run the test to make sure all is OK, then try adding the rest.

Reminder: Did you update your git repository?

4.7 Testing *predictClass()*

Now that all the other methods have been tested, we can now look at the method that predicts the classification. This is done in two parts, first with the stage2Average set to zero, and the second will be with various values for the stage 2 average. Remember that if there is a value for stage2Average, it will weight the stage 2 and final stage marks 30%/70%, otherwise the stage 3 mark only will be used.

Without the stage 2 average:

A list of test cases to test all the predictions has been prepared. In the zip file you will find an Excel spreadsheet (**TestCases7-9.xlsx**) set up with some values. Open this spreadsheet (**WithoutStage2** page) and review the values given.

The code is shown below. It uses arrays to hold the test data and then a for loop to run the tests.

```
System.out.println("predictClass");

// Array to hold the stage 3 marks
double[] marks = {0.00, 50.00, 50.00, 100.00, 39.99, 40.0,
    49.99, 50.0, 59.99, 60.0, 69.99, 70.0, 99.99, 35.67,
    44.22, 56.39, 64.00, 76.80};
// Array of corresponding classifications with no stage 2 marks
String[] expResult1 = {"No marks!", "Lower 2nd",
    "Lower 2nd", "1st", "FAIL", "3rd", "3rd", "Lower 2nd",
    "Lower 2nd", "Upper 2nd", "Upper 2nd", "1st", "1st",
    "FAIL", "3rd", "Lower 2nd", "Upper 2nd", "1st"};

// Run tests with no stage 2 average
for (int count = 0; count < marks.length; count++) {
    full.resetValues();
    full.addModuleMark(120, marks[count]);
    assertEquals("120 credits, mark = " + marks[count], expResult1[count],
            full.predictClass());
}
```

Note the way in which an array is declared and initialised.

Iterates through the arrays, using the marks and the expected results

The test method uses a *for* loop to iterate through the tests - this makes it easier to add further tests if necessary, and simplifies the coding of the tests themselves (remember - code re-use is one of the areas we are aiming for!).

Note the use of the *length* keyword that determines the number of elements in the array. Using this means that you don't have to store the array size anywhere, and if you add or remove elements, the change in size is automatically catered for.

Now run the test, correcting any bugs that may be found.

With the stage 2 average:

Now the test needs to be extended so that the stage 2 average is taken into account. In the spreadsheet, the **With Stage 2** page shows values for including the stage 2 mark. The same stage 3 marks are used.

Add a new array of doubles called **stage2** and initialise it with the values in the **Stage 2 Average** column in the spreadsheet. Ensure the values are correct and in the right order, otherwise the test will fail unnecessarily.

Then add an array of doubles called **expResult2** and initialise it with the values in the **Expected results** column in the spreadsheet. Again ensure the values are correct and in the right order.

Run the test.

Insufficient credits

There is one more test. If there are less than 120 credits, the classification cannot be predicted. Within the *predictClass()* method a message is output if this is the case. Test the result of using the *empty* object and the *halfFull* object.

- Reminder: Did you update your git repository?

4.8 A Final Test

We have been using *addModuleMarks(int, double)* to set the data, but this isn't normally the way in which this class would be used. It would be usual to create an empty object and then add marks in 10 or 20 credits at a time, as they became available. In this exercise you will add a new method to the test class to do just this.

Note that this also demonstrates that you can create your own test methods as well as using the generated methods, and that you don't have to just test individual methods.

- Place the cursor just below the final method in the test class.
- Add the following method stub:

```
@Test
public void testFullOperation() {

}
```

For the body of the method, follow the instructions below.

- Declare and initialise input values and a new **StageResults** object:

```
int[] credits = {10, 10, 10, 20, 20, 40, 10};
```

```
double[] marks = {60.6, 44.45, 80.0, 56.99, 62.3, 68.4, 59.11};
double stage2 = 61.2;

StageResults finalTest = new StageResults();
```

- Now add the code to add in the module marks and set the stage 2 average:

```
// Add in the module marks and set the stage 2 average
for (int count = 0; count < credits.length; count ++)
   finalTest.addModuleMark(credits[count], marks[count]);
finalTest.setStage2Average(stage2);
```

- … then we can test the average and the predicted class:

```
// Test the results
assertEquals("stage 3 average", 63.03,
        finalTest.calculateAverageSoFar(), 0.0);
assertEquals("predicted class", "Upper 2nd",
        finalTest.predictClass());
```

- Run the test – it should pass. If not check your code!

That's the end of this part, so you can close the project.

- Reminder: Did you update your git repository?

5 Review

Now that you have completed this part of the tutorial, this is a review of the use of JUnit and its benefits.

- JUnit is a **framework** that allows you to create tests for classes. NetBeans happens to have the functionality to generate the outline of test classes for you based on your entity classes. You then have to add the detail of the code.
- If you use this approach, you should find that testing is **repeatable**. If you rely on a console application with user input and println statements, it is tedious to re-enter all the test cases and you can (and will) get lazy!
- You don't have to **know** what results to check against which set of input, as the framework does that for you.
- **Regression testing** is easier. This is what you need to do once you have modified a class, to check that your modifications have not upset existing code. You just add new test cases for the new code.

JUnit/NetBeans does this:

- generates a test method for each of your methods in the class. The name of the method is *test* followed by the name of the method being tested.
- uses *assertEquals(…)* to generate PASS or FAIL messages depending whether the assertion is true or false. There are various signatures available to allow for int, double and String values. See the lecture slides for details.
- allows you to use @Before, @After and @Test annotations to define the use of the methods in test classes.

6 A Further Exercise

This exercise is based on the *BankAccount* class in the provided BankAccountTesing project. The project provided includes some bugs, you need to create a set of JUnit tests to test the class and then remove the bugs.

When an account is created, you need to supply the account holder's name in the constructor; the initial balance is set by the constructor to a default of 100, and the overdraft to 500.

Before you start, write down some test cases and the expected results. You need to create some objects and set up some tests for:

- **depositMoney(int)** ~ check that the correct amount is added to the balance
- **WithdrawalIsOK(int)** ~ check that this returns the correct amount in a variety of cases, e.g. when a withdrawal is less than the balance, is between the balance and the overall credit (i.e. balance + overdraft), and over the limit. Boundary cases should also be tested.
- **withdrawMoney(int)** ~ check that withdrawals result in the correct balance.

To create the test class, you will need to right-click **BankAccount**, and then select **Tools > Create JUnit Tests**. Do not include testing for the basic get/set methods.

Hints

Code for the Marks class

```
public class Marks {
    private int coursework;
    private int exam;

    public Marks() {
        coursework = -1;
        exam = -1;
    }

    public Marks(int coursework, int exam) {
        this.coursework = coursework;
        this.exam = exam;
    }

    public void setCoursework(int coursework) {
        this.coursework = coursework;
    }

    public void setExam(int exam) {
        this.exam = exam;
    }

    public int getCoursework() {
        return coursework;
    }

    public int getExam() {
        return exam;
    }

    @Override
    public String toString() {
        return "Marks{" + "coursework=" + coursework + ", exam="
              + exam + '}';
    }

    public double getResult(int cWeight, int eWeight) {
        double result;

        if (coursework == -1 || exam == -1)
            result = -1;
        else{
            result = (coursework * cWeight / 100.0) +
                    (exam * eWeight /100.0);
        }
```

```
        return result;
    }
}
```

Testing the setExam method

```
System.out.println("Testing setExam(40)");
System.out.println("Result should be coursework = 60, exam = 40");
marksObject.setExam(40);
System.out.println(marksObject.toString());
System.out.println("\nTesting getResult with marks of 60 and 40");
System.out.println("Result should be 50.0");
System.out.println(marksObject.getResult(50, 50));
```

Testing getResult with non-integer response

```
System.out.println("\nTesting getResult with marks of 51 and 66");
System.out.println("Result should be 58.5");
marksObject.setCoursework(51);
marksObject.setExam(66);
System.out.println(marksObject.getResult(50, 50));
```

Testing addModuleMark(int, double)

```
// Add a 10 credit module, mark 70%
empty.addModuleMark(10, 70.0);
assertEquals("Credits should be 10", 10, empty.getTotalCredits());
assertEquals("Marks should be 70", 70.0, empty.getTotalMarks(), 0.0);

// Add a 20 credit module, mark 40%
empty.addModuleMark(20, 40.0);
assertEquals("Credits should be 30", 30, empty.getTotalCredits());
assertEquals("Marks should be 150", 150.0, empty.getTotalMarks(), 0.0);

// Add a 40 credit module, mark 80%
empty.addModuleMark(40, 80.0);
assertEquals("Credits should be 70", 70, empty.getTotalCredits());
assertEquals("Marks should be 470", 470.0, empty.getTotalMarks(), 0.0);

// Restore the object to its empty state
empty.resetValues();
```

Testing calculateAverageSoFar()

```
// Test with no credits and no marks
assertEquals("empty", 0.0, empty.calculateAverageSoFar(), 0.0);

// Test with 120 credits all at 50%
assertEquals("full @ 50%", 50.0, full.calculateAverageSoFar(), 0.0);

// Test with 120 credits all at 100%
full.resetValues();
```

```
full.addModuleMark(120, 100.0);
assertEquals("full @ 100%", 100.0, full.calculateAverageSoFar(), 0.0);

// Test with 120 credits all at 43.92%
full.resetValues();
full.addModuleMark(120, 43.92);
assertEquals("full @ 100%", 43.92, full.calculateAverageSoFar(), 0.0);
full.resetValues();
full.addModuleMark(120, 50.0);

// Test with 60 credits at 50%
assertEquals("60 credits @ 50%", 50.0, halfFull.calculateAverageSoFar(),
0.0);

// Test with 60 credits at 100%
halfFull.resetValues();
halfFull.addModuleMark(60, 100.0);
assertEquals("60 credits @ 100%", 100.0, halfFull.calculateAverageSoFar(),
0.0);

// Test with 60 credits at 64.77%
halfFull.resetValues();
halfFull.addModuleMark(60, 64.77);
assertEquals("60 credits @ 64.77%", 64.77, halfFull.calculateAverageSoFar(),
0.0);
halfFull.resetValues();
halfFull.addModuleMark(60, 50.0);
```

Testing insufficient credits

```
assertEquals("No prediction for 60, i.e. < 120 credits",
        "Insufficient credits", halfFull.predictClass());
assertEquals("No prediction for 0, i.e. < 120 credits",
        "Insufficient credits", empty.predictClass());
```