

Unit Testing

License



This work by Duncan Mullier at Leeds Beckett University is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

Contents

[Unit Testing](#)

[License](#)

[Contents](#)

[1 Introduction](#)

[2 Setting up JUnit](#)

[3 Creating Tests](#)

[4 JUnit Test methods](#)

[5 Testing some methods](#)

[6 The Tasks](#)

The labs are written to be informative and, in order to aid clarity, instructions that you should actually execute are generally **written in this colour**. Instructions asking you to record something in your logbook are generally written **in this colour**.

1 Introduction

The distinction between unit testing and running ad-hoc tests is:

- you can easily run your unit test again and again (this is also called regression testing)
- you have a "framework" that facilitates the testing--it might actually run the test automatically whenever you build (i.e. compile) or deploy your application

When unit testing is implemented the right way it helps the programmers to become more productive, while at the same time increasing the quality of the developed code. It's important to realise that unit testing should be part of the development process, and that code must be designed so it *can* be tested. Actually the trend today is to write the unit test code *before* the code to be tested (Test Driven Development), to put focus on the interface and behaviour of your classes. This is not the case in this exercise as the code is given to you.

2 Setting up JUnit

You may complete this tutorial using Eclipse or another IDE. Eclipse makes the process fairly simple. If necessary you could use Maven. See below.

Eclipse will put the necessary jars in your build path when you create a JUnit test, see below.

[For IntelliJ see this link.](#) (Follow it up to and including Add Dependency, you probably should also go to the Advanced setting of the project and make the groupId something to do with you (I made mine uk.ac.leedsbeckett.mullier.sesc).

Otherwise it is probably simpler to use Maven to install all the dependent jar files.

If for your system you need to manually install JUnit then see <http://www.junit.org/>. It isn't a simple task as there are quite a few dependencies and it may be easier to take the Maven option below.

2.1 Setting up the Example project

There is an example project for you to test. It is very simple and has a part implementation of some software to store information about a league of cycle racing events.

Add a package in your src/main/java directory and call it **uk.ac.leedsbeckett.mullier.sesc** (you can call it what you like but you'll have to change the package statement in all of the example java files below).

Now you can add the example code. It's probably easier to manually create the files and copy and paste the code in.

[Use the classes below:](#)

[Competitor.java](#) - this class represents a competitor in a cycling league

[CycleLeague.java](#) - this represents the league itself

[Award.java](#) - represents trophies and awards

[League.java](#) - **Note, this goes in the default package.** This is main program class and has some ad-hoc (not unit) tests in it.

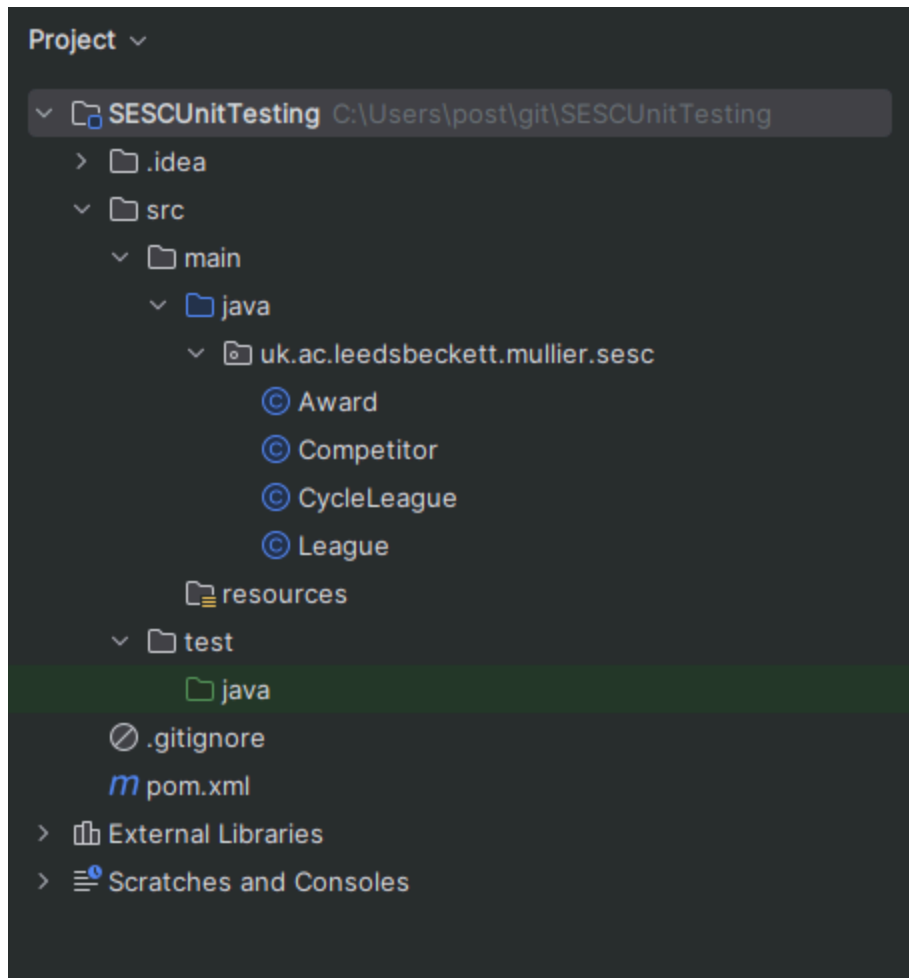
You'll get syntax errors until you've made all the files.

The program is written in a fairly clear way but it isn't well-commented.

Next create a directory to put your tests in.

You could just have them in the projects src directory but it is clearer to separate them out. If using Maven (see below) it will have already created a tests directory. It may have also created App.java in the src directory and AppTest.java in the test directory. Delete these.

If you are not using Maven then manually create a tests folder.



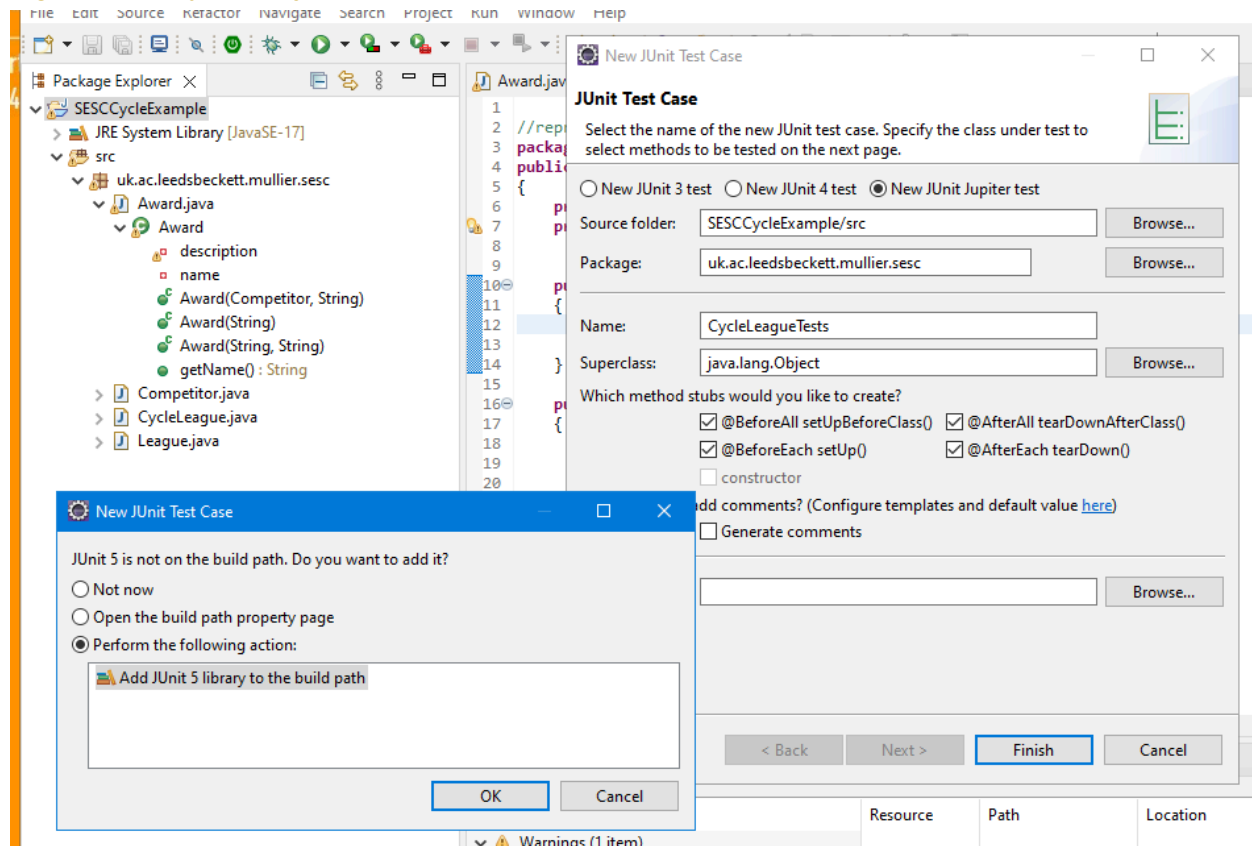
In Eclipse Right click on your project and select “properties”.
Select “Java Build Path” then “Source” then “Add Folder”->”New Folder”
create a folder called “tests” in your project.

2.2 Using Eclipse and IntelliJ

2.2.1 Eclipse

Eclipse will set up JUnit for you. But first it is clearer to put the tests in their own directory.

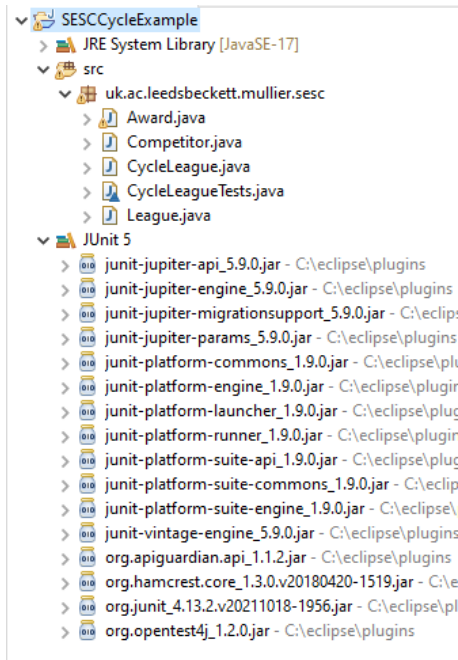
Right click on your project and select new->JUnit Test Case



We are going to use the latest JUnit Jupiter test framework, so select "New JUnit Jupiter test". Also select all of the "which method stubs would you like to create".

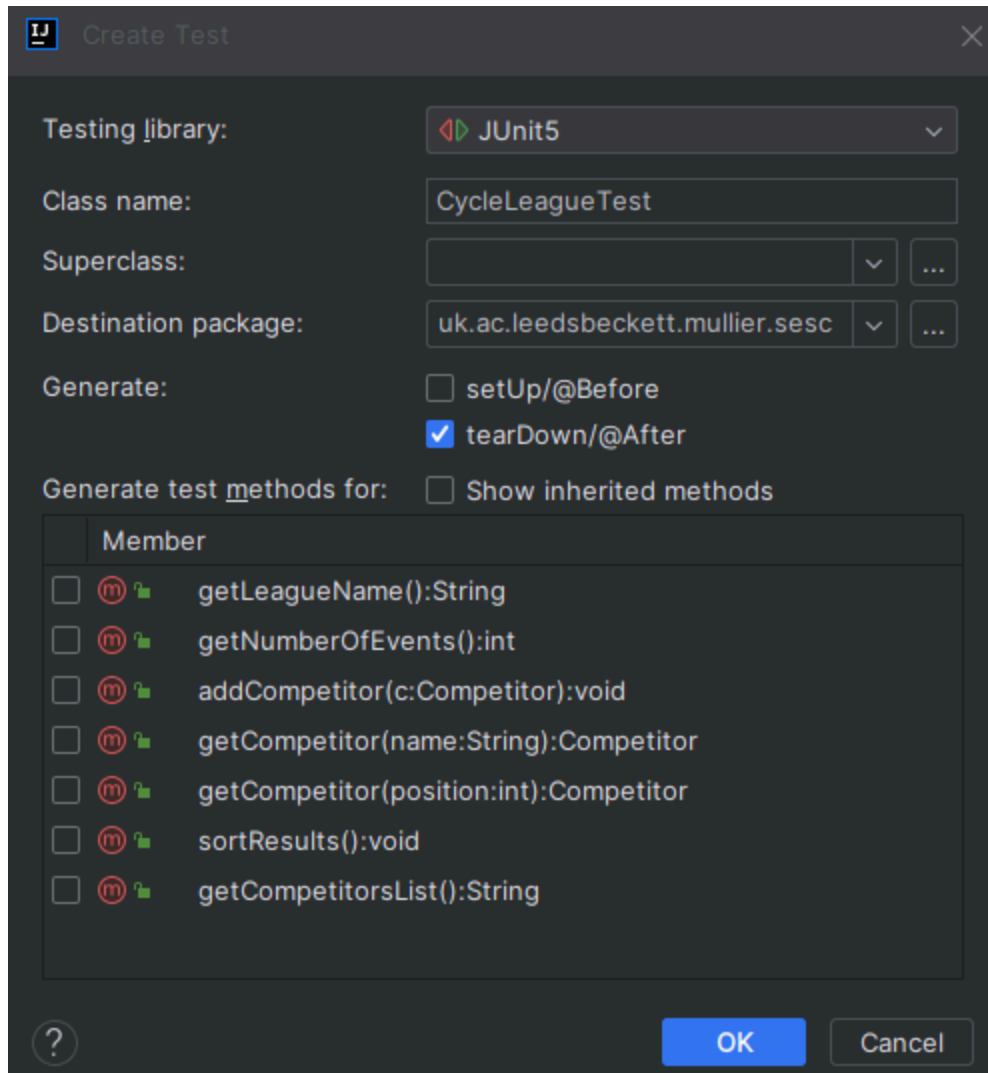
Clicking **finish** will bring up a dialogue asking if you want to add JUnit 5 to the library build path. Select **“ok”**.

Once you’ve done it you’ll see a new folder called **“JUnit 5”** has been added to your project and if you expand it you’ll see all the jar file dependencies.



2.2.3 IntelliJ

In **IntelliJ** right click on the declaration of your class (public class Award) and select **“Show Context Actions->Create Test”**.



2.3.4 Generic Testing

Here I'm creating a generic test class called `CycleLeagueTests(.java)`. I'm not actually going to use this when we get going in section 3. I'm just showing how to create a test before we get going with actually testing this code. By generic I mean that I've created one test case here, which we might use to test the entire application (`CycleLeagueTests`) but for Unit Testing we should test each Unit individually. In this case that would mean creating a test for each of the separate java classes, which is what we'll do in section 3. For now we just want to see how to create a test.

2.3 Using Maven

We could also use Maven or Gradle to handle the JUnit dependency. To be honest it is probably overkill for a project with only one dependency but if we are writing a project with many dependencies that needs Maven or Gradle then we would handle its dependency with Maven or

Gradle. To create a Maven project create a Maven Project and make it a maven-archetype-quickstart project. The project ID is the name of your project (I've called mine SESCUnitTesting).

Add the following to the pom.xml file. (See the [Maven Tutorial](#) for a refresher on Maven)..

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.9.2</version>
  <scope>test</scope>
</dependency>
```

The <! Tag is a comment tag showing where it came from for reference.

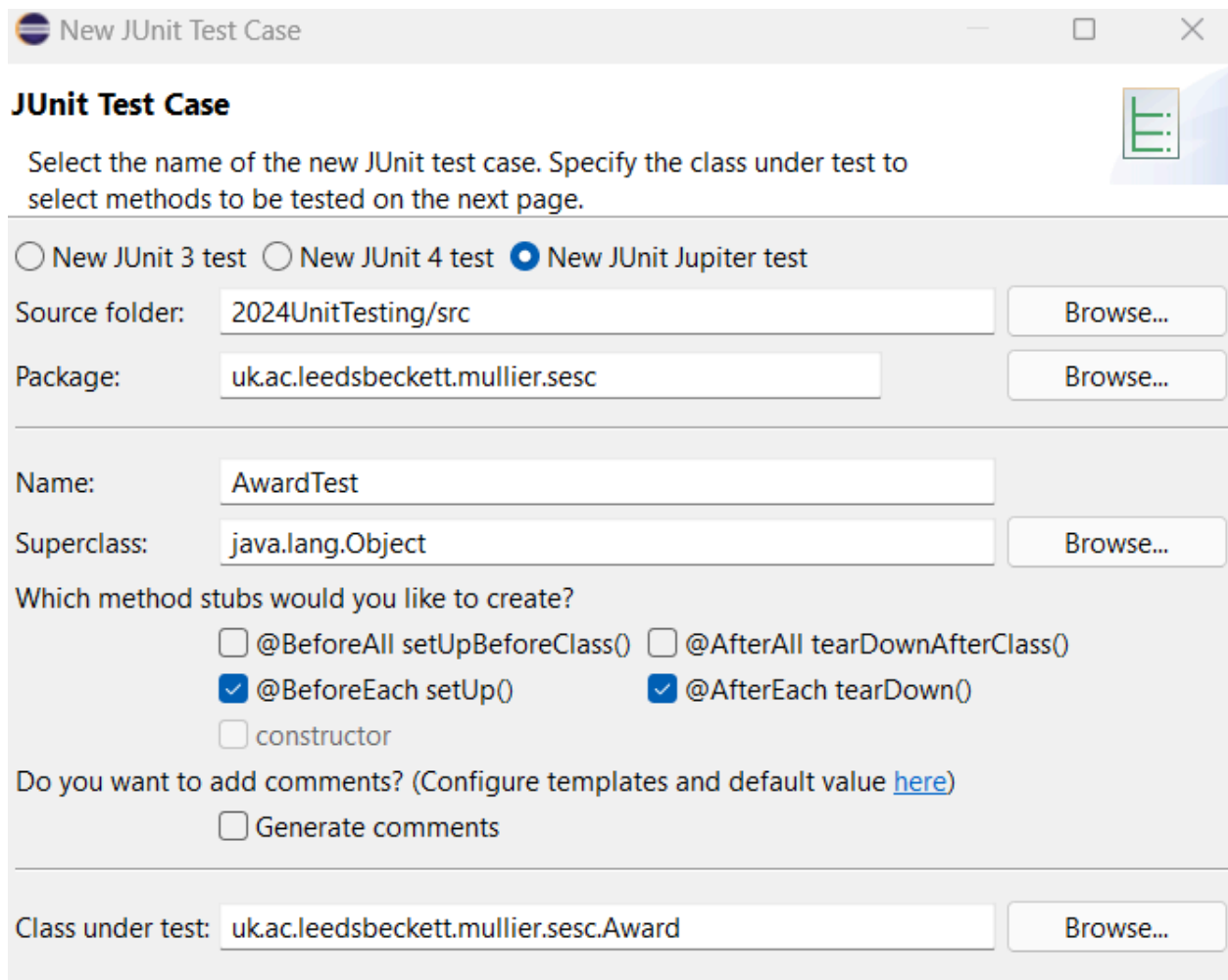
The above comes from the Junit5 webpage, section [4.2.2 Maven->Configuring Test Engines](#)

Once done build the project by right clicking on the pom.xml file and selecting run as->Maven Install.

3 Creating Tests

You can delete the CycleLeagueTests created above if you like, because we are going to create individual unit tests.

Now you can create a JUnit test by right clicking one of your classes and selecting “new->JUnit Test Case”, or in IntelliJ right click on the class definition in the source code and select “Show Context Actions”. Make sure you select the “test” directory that you created (IntelliJ will do this automatically). Click “next” and select the methods that you want to test. If Eclipse asks you if you want to add JUnit to your build path then say “yes”. Remember that all this clicking in Eclipse/IntelliJ is just to use a convenient wizard. You can easily do it manually or use the wizard in whatever IDE you are using.



JUnit Test Case

Select the name of the new JUnit test case. Specify the class under test to select methods to be tested on the next page.

☐ New JUnit 3 test ☐ New JUnit 4 test ☒ New JUnit Jupiter test

Source folder:

Package:

Name:


Superclass:

Which method stubs would you like to create?

☐ @BeforeAll setUpBeforeClass() ☐ @AfterAll tearDownAfterClass()
☒ @BeforeEach setUp() ☒ @AfterEach tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

Class under test:

 Create Test ✕

Testing library: JUnit5 ▼









Class name:

Superclass: ▼ ...

Destination package: ▼ ...

Generate: ☒ setUp/@Before
☒ tearDown/@After

Generate test methods for: ☐ Show inherited methods

Member	
<input type="checkbox"/>	  getDescription():String
<input type="checkbox"/>	  setDescription(description:String):void
<input type="checkbox"/>	  setName(name:String):void
<input type="checkbox"/>	  getName():String

? OK Cancel

That will create a

class which should look something like this:

```
package uk.ac.leedsbeckett.mullier.sesc;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;

import static org.junit.jupiter.api.Assertions.*;

class AwardTest {

    @BeforeEach
    void setUp() {
    }

    @AfterEach
    void tearDown() {
    }
}
```

Make a note in your logbook of the stages you took to get your test set up.

If you get syntax errors on the package statement it is because your project name doesn't match mine. Just change them. If you get syntax errors on the JUnit5 @BeforeXXX tags then you did not complete the above steps correctly. Try again. (You may not have selected JUnit Jupiter tests".)

4 JUnit Test methods

You can use the following methods from JUnit to test your classes, as well as your own code.

Statement	Description
<code>fail(String)</code>	Let the method fail, might be usable to check that a certain part of the code is not reached.
<code>assertTrue(true);</code>	True
<code>assertEquals([String message], expected, actual)</code>	Test if the values are the same. Note: for arrays the reference is checked not the content of the arrays
<code>assertEquals([String message], expected, actual, tolerance)</code>	Usage for float and double; the tolerance are the number of decimals which must be the same
<code>assertNull([message], object)</code>	Checks if the object is null
<code>assertNotNull([message], object)</code>	Check if the object is not null
<code>assertSame([String], expected, actual)</code>	Check if both variables refer to the same object
<code>assertNotSame([String], expected, actual)</code>	Check that both variables refer not to the same object
<code>assertTrue([message], boolean condition)</code>	Check if the boolean condition is true.

5 Testing some methods

Now let's put in a test method to test the setting and getting of a specific result. If you haven't created a stub class for a given test method then don't worry as you can write one from scratch, but note your method must begin with the word "test" and have the `@test` decoration. The test decoration tells the IDE that it is a test and so you can execute the tests from the "Run Tests" button and get a nice graphical output of ticks and crosses.

We could make all of our tests completely stand alone or we can use the above methods to set them up. There are reasons for doing each as circumstances dictate, but generally if a set of tests can use the same setup then they should because you are cutting down on the amount of code produced.

In our case we will set up a `CycleLeague` and a competitor.

```
@BeforeEach
void setUp() throws Exception
{
    System.out.println("allocating resources");
    league = new CycleLeague("Time Trial League",18);
    c = new Competitor(league,"Test Person","Nova","Vet");
}
```

This will be called before all `@tests` and an appropriate `CycleLeague` and `Competitor` objects will be created to be used in our tests. Note here we must create an instance reference "league" and "c". We can't create them inside this method because we want to use it elsewhere and they would be out of scope.

```
private CycleLeague league;
private Competitor c;
```

For completeness we can deallocate resources after each test.

```
@AfterEach
void tearDown() throws Exception
{
    System.out.println("deallocating resources");
    //release resources
    league = null;
    c = null;
}
```

Now we can produce an actual test. Put the following code in:

```
@Test
/**
 * tests getPointsEvent(int event)
 * sets the points using setResult(int event, int points)
 * uses getPointsEvent(int event) to read the set value back
 */
void testResult()
{
    c.setResult(5, 50);
    assertEquals(c.getPointsEvent(5), 5);
}

@Test
/**tests GetPoints()
 * sets five sequential events to 10 points each using
 setResult()
 * reads the total points back using getPoints()
 */
void testGetPoints()
{
    c.setResult(10);
    c.setResult(10);
    c.setResult(10);
    c.setResult(10);
    c.setResult(10);
    assertEquals(c.getPoints(), 50);
}
```

If you get any syntax errors on the @ decorations then right click on them and it will suggest adding the correct import.

We can also put in @BeforeAll and @AfterAll decorations. Eclipse will give us this option but, for some reason IntelliJ doesn't. Add them in.

Put a System.out.println("in beforeAll"); etc in @BeforeAll and @AfterAll methods, so we will be able to see in the console.

To run the tests in Eclipse select "Run->Run As->JUnit Test".

To run the tests in IntelliJ just click run when the Test class is open.

Copy the output from the console into your logbook.

When you run the tests the following will happen:

@BeforeAll is called, which just outputs a message.

@BeforeEach is called and the test conditions are set (objects created for tests)

@Test is called, in this case our testGetName() the objects in the previous call are used.

The assertEquals() determines whether the test passes or not.

Reference "c" points to a Competitor object who's getName() method is called, if this is equal to "Test Person" then the test passes, otherwise it fails.

After the test is complete then:

@AfterEach is called and the resources are deallocated by setting the object references to null.

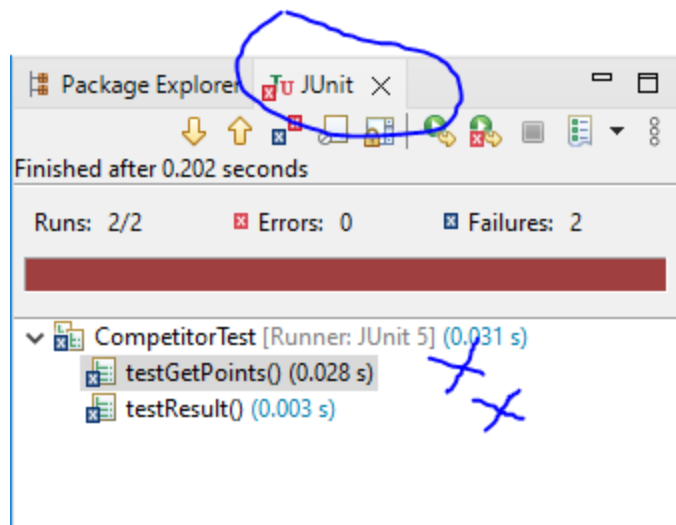
@AfterAll is called which just outputs a message.

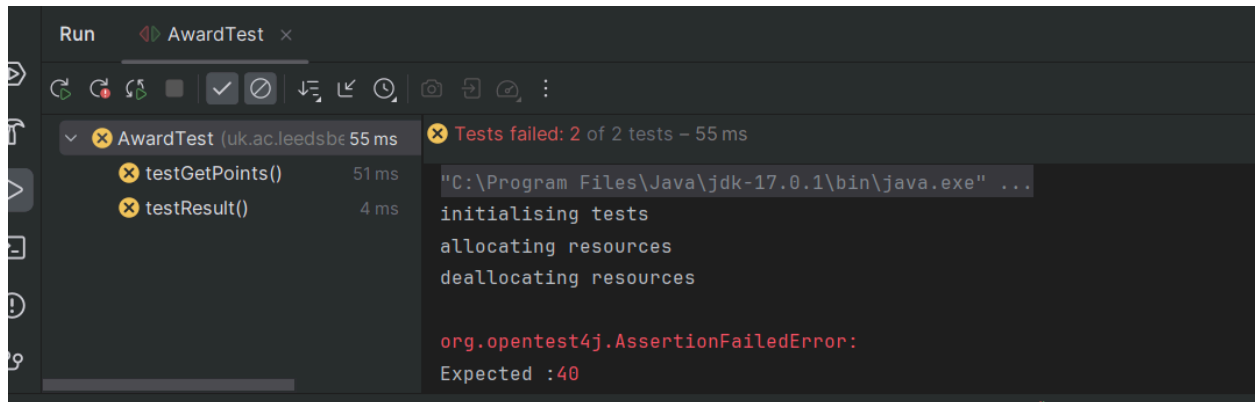
Eclipse (or IntelliJ) will show a nice little tick if your test passed.

The JUnit tab should appear. If you don't see the JUnit window select Window->Show View->JUnit

Any additional tests can be put anywhere in your code, so long as they have the @test decoration they will execute after @BeforeAll and will produce an additional call to @BeforeEach and @AfterEach.

It's failed!



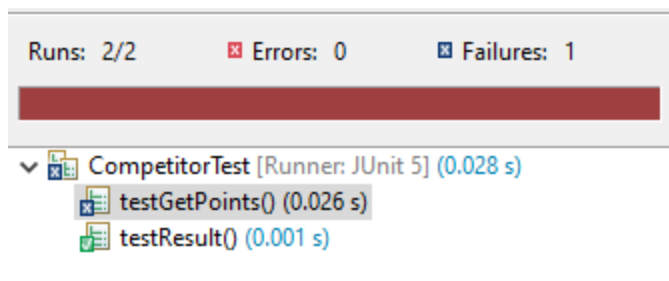


Did you sport my deliberate mistake? We know something has failed and which test method by the cross next to it. This time it is an error in my testing code as I set event 5 to 50 then tested to see if the return value was 5. Silly me.

```
assertEquals(c.getPointsEvent(5), 50);
```

assertEquals() here is taking two values and saying the test passes if they are equal. So if c.getPointsEvents(5) returns 50 it will pass and it should because that is what we set it to. If it doesn't the error is either in our test method, in the setter or in get getter. It isn't as likely to be in the test methods though as it is inherently a simple method. I.e. your tests should be clear and concise and if a test ends up being more complicated than the thing you're testing then it shouldn't be tested with unit tests.

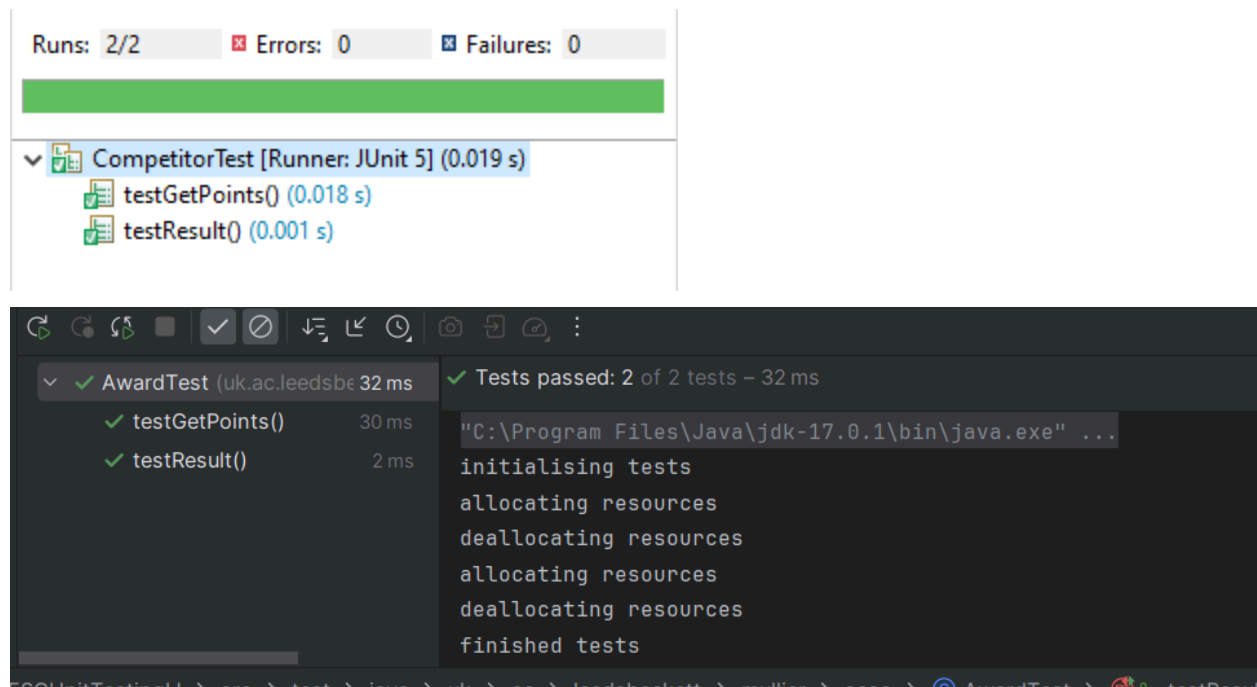
Correct it and run it again.



Test result is working but testGetPoints isn't. Strictly speaking testing is not debugging, but methodologies like Test Driven Development require that you build the tests first, hence they will always fail at first. Here we must resort to looking at the code it is testing.

Look at the getPoints() method in competitor.java and see if you can spot the error.

Correct it and run the tests again (if you really can't spot it the answer is at the bottom).



Show the steps you have taken to run a failed and passed test in your logbook.

6 The Tasks

Your task, as the tester and not the original programmer, is to write a full set of test methods for the classes CycleLeague.java and Competitor.java and Award.java. There may be one or two problems with the classes, but you'll find that out and correct them. You might also think that the code isn't very well commented and decide, for your own aid as much as anyone else's, to comment it properly. Your test methods at least should be clearly commented to show what they are supposed to be doing. You should produce tests minimally for the following:

Adding a competitor to the league and making sure that they are there.

Putting a result in and getting it out again.

Ensure that sort doesn't mess anything up.

Check that an award can be correctly assigned to a competitor.

Fix any bugs you find and document them.

Provide suitable additional tests.

Provide full details in your logbook of all your tests.

The error in Competitor.java

```
public int getPoints()
{
    int points=0;
    for(int i=1 ; i<eventsCompeted; i++)
        points = points + results[i];
    return points;
}
```

Can you spot it? It isn't getting the first (zero) result. It should be 0.

References

<https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api/5.9.2>

Projects

IntelliJ with local Jar files

<https://github.com/LBU-SESC/SESCJUnitLocalIJ.git>

To manually add the jar files to your own project. Create a new directory called "jar" in your project. Copy the jar files in this project to that directory (download them) then go to the main menu->Project Structure->Modules->Dependencies click + the Jars and Directories and add your jar directory.

IntelliJ with Maven

<https://github.com/LBU-SESC/JUnitTestExampleIJ.git>

Eclipse with local jar files stored in your local Eclipse instal

<https://github.com/LBU-SESC/JUnitTestExample.git>