**Topic 10: Introduction to Unit Testing with JUnit**

These notes will introduce you to software testing (specifically, unit testing) and how to accomplish it in Java using the JUnit framework.

**Testing general intro**

Obviously it is important to test software before releasing it, to iron out bugs. Software testing can be done in an informal, ad-hoc way, however the disadvantage of this is that the developer is likely to miss out testing crucial functionality. The robustness of the software can be enhanced by taking a more formal approach to testing, by drawing up a ***test plan*** documenting all tests and the expected and actual output, as well as performing a series of ***unit tests*** designed to test different parts of the system.

**Unit testing**

A ***unit test*** is designed to test one small part of the system in isolation, such as a method. Unit tests are written to test different outcomes of a method. For example, an Event class (used in the live music venue application) might have a book() method which takes the number of tickets to book as a parameter. This could have three outcomes:

* The event is booked successfully;
* There are insufficient tickets available;
* The number of tickets is invalid, i.e. zero or a negative number.

We would create unit tests to test each of the three possible outcomes and check that the expected behaviour does indeed occur in each case.

public class Event

{

int tickets;

// ... rest of class omitted

public boolean book (int amount)

{

if(tickets < amount)

{

return false;

}

else if (amount <= 0)

{

return false;

}

tickets -= amount;

return true;

}

public int getTickets()

{

return tickets;

}

}

Unit tests for this book() method could involve:

* Making a valid booking and checking that true is returned;
* Making a valid booking and checking that the amount is reduced by the expected amount;
* Supplying an invalid amount (0 or less), checking that false is returned;
* Supplying an invalid amount (0 or less), checking that the number of tickets does not change;
* Supplying an amount greater than the number of tickets, and checking that false is returned;
* Supplying an amount greater than the number of tickets and checking that the number of tickets does not change.

**JUnit - A Unit Testing Framework**

In Java, unit testing is made straightforward by the open-source unit testing framework [JUnit](http://www.junit.org/). This can be used from Netbeans by adding the required libraries as a dependency to your pom.xml within the <dependencies> section. We are using JUnit 5, as this is the latest version. It requires Java 8 or higher. If you are having problems with JUnit 5, there are also some (older) notes for [JUnit 4](https://nwcourses.github.io/COM528/junit4.html) here.

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>5.7.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.7.2</version>

<scope>test</scope>

</dependency>

<!-- This is needed only if you are doing parameterised tests (see below) -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-params</artifactId>

<version>5.7.2</version>

<scope>test</scope>

</dependency>

To use JUnit 5, you also need to update the Maven Surefire plugin, which is used to automate your tests, to at least version 2.22. Add this to your pom.xml below the dependencies.

<build>

<plugins>

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.0</version>

<configuration>

<includes>

<include>\*\*/\*Tests.java,\*\*/\*Test.java</include>

</includes>

</configuration>

</plugin>

</plugins>

</build>

There is a good tutorial [at Vogella](http://www.vogella.com/articles/JUnit/article.html) which was partly used to research these notes.

**Example**

Imagine you have an Event class above. To use JUnit, we have to create a ***test class*** with a series of unit tests. Here is an example of a series of unit tests that you would run with JUnit on the Item class above:

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

import org.junit.jupiter.api.Test;

public class EventTest {

// Test that selling one ticket returns true if there are enough tickets

@Test

public void testSellOneTicketSuccess()

{

// Arguments of Event: name, number of tickets

Event e = new Event("Postmodern Doom", 100);

assertTrue( e.book(1) );

}

// Test that selling one ticket reduces the stock appropriately if there are enough tickets

@Test

public void testSellOneTicketReducesStock()

{

Event e = new Event("Postmodern Doom", 100);

e.book(1);

assertEquals( e.getTickets(), 99 );

}

// Test that selling one ticket returns false if there are not enough tickets

@Test

public void testSellOneTicketNoAvailability()

{

Event e = new Event("Postmodern Doom", 0);

assertFalse( e.book(1) );

}

// Test that selling one ticket returns false if the quantity is invalid

@Test

public void testSellOneTicketInvalidQuantity()

{

Event e = new Event("Postmodern Doom", 100);

assertFalse( e.book(-1) );

}

// Test that selling one ticket does not reduce the stock if there are not enough tickets

@Test

public void testSellOneTicketNoAvailabilityDoesNotChangeStock()

{

Event e = new Event("Postmodern Doom", 0);

e.book(1);

assertEquals( e.getTickets(), 0 );

}

// Test that selling one ticket does not change the stock if the quantity is invalid

@Test

public void testSellOneTicketInvalidQuantityDoesNotChangeStock()

{

Event e = new Event("Postmodern Doom", 100);

e.book(-1);

assertEquals( e.getTickets(), 100 );

}

@Test

public void testSellTwoTicketsReducesStock()

{

Event e = new Event("Postmodern Doom", 100);

e.book(2);

assertEquals( e.getTickets(), 98 );

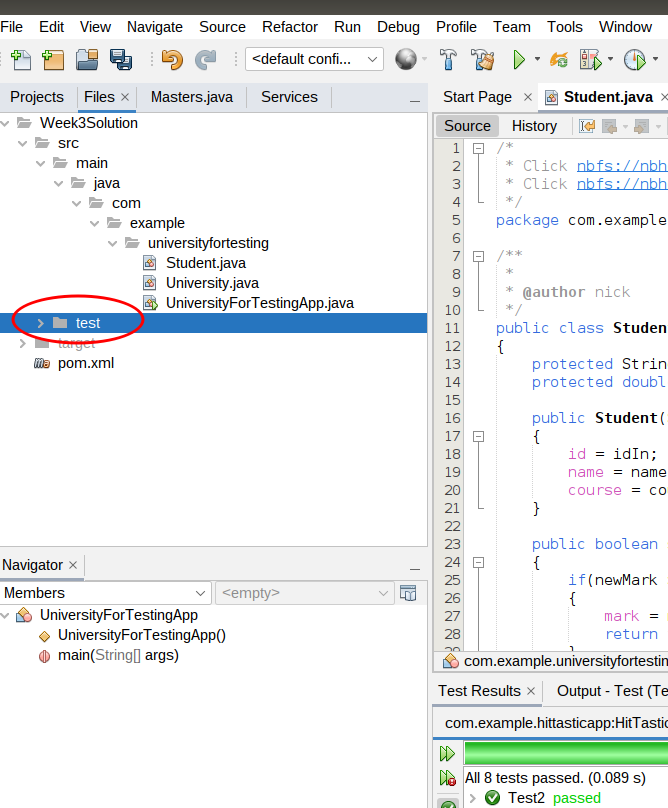
}

}

Looking at this in more detail:

* Note how we have a series of test methods. Note how each is preceded with the annotation ***@Test***. This informs JUnit that the method coming up is a test that it should run. (This is actually Java syntax, not specifically JUnit; for more on Java annotations see [here](http://www.vogella.com/articles/JavaAnnotations/article.html)).
* Note how each method ***creates an instance of the class under test*** (Item here), ***performs an operation on it*** (e.g. sells it) and ***tests whether a condition has been met*** with an **assertion**. Assertions include assertTrue() (tests whether a method returns true), assertFalse() (tests whether a method returns false), or assertEquals() (which tests whether a return value is equal to specified value.

**Adding tests in Netbeans**

Test classes should be placed in the test folder within your project. Right-click on the test folder, as shown below, to add a new test class:  


**Parameterised Tests**

Often, tests differ only in the parameter we are supplying to the method. You can see this above in the tests within EventTest; most only differ by the amount of tickets we are supplying. Clearly this is rather inefficient. It would be better if we could run a single test with one or more ***parameters*** - and luckily we can. The example below has a single test, testSellMultipleValues() and runs it three times, passing the parameters -1, then 1, then 2 to it:

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

import org.junit.jupiter.api.Test;

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.MethodSource;

public class ParamItemTest {

public static int[][] testData()

{

int[][] data = { { -1 } , { 1 } , { 2 } } ;

return data;

}

@ParameterizedTest

@MethodSource(value="testData")

public void testSellMultipleValues(int[] data)

{

int maxTickets = 100;

Event event = new Event("Test event", maxTickets);

int expectedTickets = data[0] > 0 ? maxTickets - data[0] : maxTickets;

event.sell(data[0]);

assertEquals( event.getTickets(), expectedTickets );

}

}

Note the following:

* Note the testData() method. This provides the parameters to send to the test. It creates and returns a two-dimensional int array (int[][]) of the parameters to send to the test.
* @Parameters
* public static int[][] testData()
* {
* int[][] data = { { -1 } , { 1 } , { 2 } } ;
* return data;
* }

This method returns an array of parameters to test: the test method will be run with each parameter in the array in turn. The data: int[][] data = { { -1 } , { 1 } , { 2 } } ; is a ***two-dimensional array*** of parameters. Each member of the array ***data*** corresponds to the parameters to test each time, but each member is ***itself an array*** so that we can send ***multiple parameters*** to the test. Here, however, we're only sending one parameter to the test so each member of testData is an array with one member - the parameter to test.

* We then come to the test itself, the testSellMultipleValues() method. This is annotated with @ParameterizedTest to indicate that it's a parameterised test, and another annotation @MethodSource to indicate where the test data is coming from.

@MethodSource(value="testData")

indicates the data is coming from a method called testData(). The current item from the test data is passed to the testSellMultipleValues() method; it will be an int array (int[]) containing one single value.

* The test method creates an Event with 100 tickets, and then tries to sell the number of tickets defined by the current parameter (i.e. -1, 1 or 2, with a different value for each run of the test). We then work out the expected number of tickets after the sale. This will either be:
  + the maximum tickets minus the parameter (if the parameter is 1 or more);
  + or the maximum tickets (if the parameter is less than 1), because numbers of tickets less than 1 should be treated as invalid and thus the number of tickets after the attempted sale will not change.

Note how we do this with the ***ternary operator*** (?), which is a short-hand way of doing a conditional statement:

int expectedTickets = data[0] > 0 ? maxTickets - parameter : maxTickets;

This is saying "if data[0] (i.e. the number of tickets we want to sell) is greater than 0, set expectedTickets to maxTickets - data[0], otherwise set it to maxTickets".

* We then check, with an assertion, whether the return value of getTickets() is the expected number.

**Parameterised Tests with Multiple Parameters**

The following example is a modification of the previous one, which takes ***multiple parameters*** per test run. We specify these via a ***CSV*** source this time:

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

import org.junit.jupiter.api.Test;

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.CsvSource;

public class ParamItem2Test {

@ParameterizedTest

@CsvSource({"-1,false","1,true","2,true","101,false"})

public void testSellMultipleValuesSuccess(int numberOfTickets, boolean expectedSellStatus)

{

Event event = new Event("Test event", 100);

assertEquals(event.sell(numberOfTickets), expectedSellStatus);

}

@ParameterizedTest

@CsvSource({"-1,100","1,99","2,98","101,100"})

public void testSellMultipleValuesRemainingQuantity(int numberOfTickets, int expectedRemainingTickets)

{

Event event = new Event("Test event", 100);

event.sell(numberOfTickets);

assertEquals(event.getTickets(), expectedRemainingTickets);

}

}

Note how we pass the test data to each test in ***comma-separated*** (CSV) format with @CsvSource, e.g:

@CsvSource({"-1,false","1,true","2,true","101,false"})

The CsvSource contains a set of test data to be used for each run of the test, as a CSV string. For the first test, testSellMultipleValuesSuccess(), we are testing whether the sell() method returns the expected boolean value when a particular number of copies are sold. The CSV contains the copies to be sold, and the expected boolean return value of sell() for that number of copies, in that order. The parameters to the method (numberOfTickets and expectedSellStatus) correspond to this data from the CSV, in the same order.

So here, the test will be run four times, the first time with the values of -1 and false; the second time with the values of 1 and true, and so on.

The testSellMultipleValuesRemainingQuantity() method works in a similar way, but this time we are testing that the remaining number of tickets is as we expect for each run of the test.

The test data in the CSV contains these cases:

* Selling -1 tickets. sell() should return false as -1 is an invalid quantity, and not change the remaining tickets (so this remains as 100).
* Selling 1 ticket. sell() should return true and leave 99 remaining tickets.
* Selling 2 tickets. sell() should return true and leave 98 remaining tickets.
* Selling 101 tickets. sell() should return false due to attempting to sell one more than the maximum available number of tickets, and as a result not change the remaining tickets (so this remains as 100).

Note how the two tests work. The first test tries to sell numberOfTickets tickets (the first parameter of the test) and checks that the return value of sell is equal to expectedSellStatus (the second parameter of the test). The second test tries to sell numberOfTickets tickets and checks that the number of remaining tickets is equal to expectedRemainingTickets (the second parameter of the test).

**Further example: the Venue**

This is an example of testing a more complex class (the Venue, which contains multiple Events). Not all possible tests are shown. As discussed below, you should test more complex classes, such as Venue, ***after*** the simpler classes (such as Event) are fully tested and working.

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

import org.junit.jupiter.api.Test;

public class VenueTest {

@Test

public void testAddEvent()

{

int id = 1;

Venue v=new Venue();

v.addEvent(new Event(id, "Postmodern Doom","1/4/23", 1000));

assertNotNull(v.findEventById(id));

}

@Test

public void testFindOneEventByName()

{

Venue v = new Venue();

int id = 1;

String name = "Postmodern Doom";

Event e = new Event(id, name, "1/4/23", 1000);

v.addEvent(e);

ArrayList<Event> events = v.findEventsByName(name);

assertEquals(events.size(), 1);

}

@Test

public void testFindTwoEventsByName()

{

Venue v = new Venue();

int id = 1;

String name = "Postmodern Doom";

Event e = new Event(id, name, "1/4/23", 1000);

Event e2 = new Event(id+1, name, "2/4/23", 1000);

v.addEvent(e);

v.addEvent(e2);

ArrayList<Event> events = v.findEventsByName(name);

assertEquals(events.size(), 2);

}

@Test

public void testFindNonExistentEventByName()

{

Venue v = new Venue();

int id = 1;

String name = "Postmodern Doom";

Event e = new Event(id, name, "1/4/23", 1000);

v.addEvent(e);

ArrayList<Event> events = v.findEventsByName("Metagalactic Hamstaaz");

assertEquals(events.size(), 0);

}

@Test

public void testFindNonExistentEventById()

{

Venue v = new Venue();

int id = 1;

String name = "Postmodern Doom";

Event e = new Event(id, name, "1/4/23", 1000);

v.addEvent(e);

assertNull(v.findEventById(id+1));

}

}

**General strategy for unit testing**

You should test each class as you write it. A good strategy is to test ***the simpler classes, with no dependencies on other classes, first***, as then you know that those classes are fully working. For example, you would want to test the Event class and a Booking class before testing the Venue class. After testing the classes with no dependencies, you can then test the larger and more complex classes (e.g. Venue), which ***use*** those simpler classes, to see if they work.

You should also ensure you test ***edge cases***. An edge case is a value on the boundary between two outcomes. For example, if a venue has 100 tickets, you might want to test whether 100 tickets can be booked (which should work), and whether 101 tickets can be booked (which should not). Edge cases are common places to find bugs (one example is confusion between < and <=) and ensuring they are included as data in tests means that such bugs are likely to be found.

**Exercise**

Clone this repository from GitHub:

https://github.com/nwcourses/COM528

This contains the solutions to previous exercises, but also a project called UniversityForTesting, which you should use for this exercise. It contains a modified version of the Week 3 solution. Look at the classes within the project.

Develop a series of unit tests for both the Student class and the University class, and run them.

**References**

[Lars Vogel's tutorial on JUnit](http://www.vogella.com/articles/JUnit/article.html)