Week 11 – Testing

Testing is a very important part of the software development process.

Upon creating software, it is important that we ensure said software is working correctly and doesn't crash. A broken or inaccurate program will undermine user's confidence in your software and your company

There are different stages of software testing:

- Unit Testing – tests a small unit, or part, of the software to check that it is working as expected
- Integration testing – tests that different components of the software will work with each other
- Performance testing – Ensures the software performs well, usually under strain (i.e. increased traffic).

In this session we will be focusing on **unit testing**

JUnit

JUnit is a java library that enables us to create and run unit tests for the software we develop

JUnit is not part of the standard java libraries; however, it is usually included with many of the popular IDEs, including:

- Netbeans
- Eclipse
- IntelliJ

The basic premise of JUnit is that we create a test class for each class of our application.

Let's take the Cars in a Carpark task we did previously.
With that system we will create two further classes for testing purposes:

- CarTest.java
- CarparkTest.java

Here is an example of the minimal structure for CarTest.java

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CarTest {

}
```

As you can see, thus far there is nothing particularly special about this class other than the fact it imports some JUnit libraries.

One of these libraries is "Assert" which plays a key role in this process as you will see below.

Author: Darren Cunningham
Version: 1.0

Week 11 – Testing

Test Methods and Assert

In JUnit we create a test method for each component of the class we wish to test. For example, we have a getter method in the car class that is supposed to return the cars registration number. We can create a method to test this:

```
@Test
public void testGetReg()
{

}
```

This is the same as any other method but note the @Test annotation at the top. This is important and should be included.

Now we can run a test, the way we do this is by making "**assertions**"

JUnit has numerous "assert" classes that enable us to do this.
Let's go back to our Car class example:
If I have a car with the registration number "ABCDEF" then I can assert that the return value of the getReg() method for that Car object should equal "ABCDEF"

```
@Test
public void testGetReg()
{
    Car c1 = new Car("ABCDEF");

    assertEquals("ABCDEF", c1.getReg());
}
```

Above you can see the assertEquals method. This is part of the JUnit library. It basically takes two values and checks that they match one another.

It can also take an optional string as the first parameter, should you wish to include a failure message

```
assertEquals("Incorrect Reg Num", "ABCDEF", c1.getReg());
```

Author: Darren Cunningham
Version: 1.0

There are many of these assert classes, some of the more common ones you might use include:

- assertEquals
- assertTrue
- assertFalse
- assertNull
- assertNotNull
- assertSame
- assertNotSame

Yes, you can use multiple asserts in a single test method

The CarparkTest Class

Now let's look at the test class for Carpark. This will aim to test the 3 methods of Carpark:

- addCar(Car car)
- removeCar(String reg)
- calcEmptySpaces()

Continued on the next page

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CarparkTest {

    @Test
    public void testAddCar()
    {
        Carpark cp = new Carpark();
        Car c1 = new Car("abcdef");

        assertTrue(cp.addCar(c1));

        // Testing the car park is full
        for(int i=1; i<15; i++)
        {
            cp.addCar(c1);
        }

        assertFalse("Car should be rejected", cp.addCar(c1));
    }

    @Test
    public void testRemoveCar() {
        Carpark cp = new Carpark();

        assertFalse("Car wont exist", cp.removeCar("ABCDEF"));

        cp.addCar(new Car("abcdef"));

        assertTrue(cp.removeCar("abcdef"));
    }


    @Test
    public void testCalcEmptySpaces() {
        Carpark cp = new Carpark();

        assertEquals(15, cp.calcEmptySpaces());

        cp.addCar(new Car("abcdef"));

        assertEquals(14, cp.calcEmptySpaces());

        cp.removeCar("abcdef");

        assertEquals(15, cp.calcEmptySpaces());
    }

}
```

In the above example, we have 3 test methods.
Note how we are testing the different expected outcomes.

Take testAddCar for example.
We need a CarPark object and a Car object to test this, so we create these first.

We then assert that if we add the car to the carpark then the addCar method should return true (which would make sense as the carpark is currently empty)

We then need to check that the method returns false if the carpark is full, thus we fill up the remaining spaces.
Then we assert that if another car is added to the carpark then the addCar method should return false (as the carpark is now full)

Test Suites

You will usually create multiple test classes and it can be a pain to individually run them one at a time.

JUnit allows us to create Test Suites in which you define what tests you want to run and the order you wish to run them in.
From there you can just run the test suite and it will run all your defined test classes at once:

CarsinacarparkSuite.java

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({carsinacarpark.CarTest.class,
carsinacarpark.CarparkTest.class})
public class CarsinacarparkSuite {

}
```

This will look confusing at first but note the first two lines are just importing JUnit libraries.

The next line down is an @RunWith annotation which tell the class how to behave (as a suite)

Then we have another annotation which states what test classes will run.
These will be run in order!

Then finally you have the class itself, which in the case can just be left empty.