JUnit

Contents

Overview

Testing aids in providing quality assurance to a business, and us as developers. Testing is the activity of running code under various circumstances to highlight where defects exist. We can split tests into three main categories; **Functional**, **Non-Functional** and **Maintenance** tests.

Functional	Non-Functional	Maintenance
Unit Testing	Performance	Regression
Integration Testing	Scalability	Maintenance
Smoke Testing	Usability	
User Acceptance Testing		

- **Functional**: If I click a button – does it do what it is supposed to?
- **Non-Functional**: If 10,000 people all click the button at the same time – does it respond in an acceptable time?
- **Maintenance**: Once changes have been made, do previously passed tests still pass now?

JUnit focuses on Unit tests, but can also be used for a variety of other use cases.

Adding the dependency

JUnit is available as a Maven dependency for Java projects, and can be found here.

You'll need to add the dependency within the `<dependencies>` tag of your Maven project's `pom.xml`:

```xml
<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

(*note: you will need to convert any Java project to a Maven project first, as this will generate the* `pom.xml` *for you.*)

Tutorial

Annotations

@Test

The `@Test` annotation denotes that the method below it should be ran as part of the JUnit tests.
A timeout can be added, forcing the test to fail if it doesn't complete within a certain time period.
Expected exceptions can be handled, allowing the test to pass when the exception is thrown.

```java
@Test
public void test1() {
    fail("Fail");
}

@Test(timeout = 1000)
public void test2() {
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

@Test(expected = ArithmeticException.class)
public void test3() {
    throw new ArithmeticException();
}
```

@BeforeClass

The `@BeforeClass` annotation precedes the method that is to be executed, only once, before the tests within that class run.
It is typically used to set up the test environment.

For example: When accessing a database, you only want to connect to the database once, before all testing, instead of connecting to it for every test.

The method this annotation is assigned to should be named `setup()`:

```
@BeforeClass
public static void setup() {
    System.out.println("Before class");
}

@Test
public void test1() {
    System.out.println("Test 1");
}

@Test
public void test2() {
    System.out.println("Test 2");
}
```

@AfterClass

The `@AfterClass` annotation precedes the method that is to be executed, only once, after the tests within that class run. It is typically used to clean the test environment.

Taking the database connection example from earlier, here would be the best place to put the database disconnection functionality, since it will be called once every test has completed.

The method this annotation is assigned to should be named `teardown()`:

```
@Test
public void test1() {
    System.out.println("Test 1");
}

@Test
public void test2() {
    System.out.println("Test 2");
}

@AfterClass
public static void teardown() {
    System.out.println("After class");
}
```

@Before

A method annotated with the `@Before` annotation is executed before each `@Test` method.
If every `@Test` method starts with the same few lines of code, we can refactor them into one `@Before` method.

This would be useful, for instance, for when we need to load data into a database to test different commands.

```
@Before
public void init() {
    System.out.println("Before test");
}

@Test
public void test1() {
    System.out.println("Test 1");
}

@Test
public void test2() {
    System.out.println("Test 2");
}
```

@After

A method annotated with the `@After` annotation is executed after each `@Test` method, irrespective of whether the test passed.

It is typically used for cleaning up any data that was used within the testing environment.

- Once the test has completed, we want to reset the data to its previous state.
- When the `@Before` method executes and loads the data into the environment, this ensures that there isn't duplicate data in any tables used for the test.

```
@Test
public void test1() {
    System.out.println("Test 1");
}

@Test
public void test2() {
    System.out.println("Test 2");
}

@After
public void reset() {
    System.out.println("After test");
}
```

@Ignore

The `@Ignore` annotation is used to skip a particular method.
It will only work in conjunction with another annotation - `@BeforeClass`, `@Before`, `@Test`, `@After`, or `@AfterClass`.
It does not matter whether the `@Ignore` annotation is listed first or second.

It is typically used to skip a method/test that does not need to be executed.

Although this isn't very efficient when toggling many tests to run or not, it is a simple method of skipping tests.

```
@Test
@Ignore
public void method3() {
    System.out.println("Test 1");
}

@Ignore
@Test
public void method4() {
    System.out.println("Test 2");
}
```

Other Annotations

The remaining annotations are used with **Test Runners** and **Suites**:

- A *suite* is a collection of test classes.
- A *test runner* can execute many suites.

Annotation	
`@RunWith`	Allows a class to use JUnit Core runners; replaces default runner class
`@Parameters`	Signifies the parameter dataset to use for the test(s)
`@Category`	Used to add metadata for a test
`@IncludeCategory`	Used to include tests with specific metadata

Annotation	
`@ExcludeCategory`	Used to exclude tests with specific metadata

Execution Order

```
@BeforeClass
public static void setup() {
    System.out.println("Before class");
}

@Before
public void init() {
    System.out.println("Before test");
}

@Test
public void test1() {
    System.out.println("Test 1");
}

@Test
public void test2() {
    System.out.println("Test 2");
}

@After
public void finalise() {
    System.out.println("After test");
}

@AfterClass
public static void teardown() {
    System.out.println("After class");
}
```

If we execute the above tests, The following will be printed to the console:

```
Before class
Before test
Test 1
After test
Before test
Test 2
After test
After class
```

Asserts

With JUnit, we establish whether a test has passed or failed by asserting certain states of the code.

The most common assert used to pass a test is the `assertEquals()` method.

Only one assert method should be used per test.
An assert method should be the last line in your test method.

assertEquals()

`assertEquals()` is the most common used assert method to evaluate a test's pass or fail condition.

The message should be an informative description of what is to be expected.
This message is shown on a test failure in the console.

Both the expected value and the actual value need to be equal in both type and value for the test to pass, otherwise, the test will fail.

The basic structure for `assertEquals()` is:

```
assertEquals(<Message to be displayed upon failure> , <Expected value> , <Actual
value>);
```

```
@Test
public void playerWinsTest() {
    BlackJack blackJack = new BlackJack();
    assertEquals("Expected: Player wins with 21", 21, blackJack.play(21, 18));
}
```

assertEquals() Overloads

There are many overloaded variants of the `assertEquals()` method. They are as follows:

- `assertEquals(double expected, double actual, double delta);`
- `assertEquals(long expected, long actual);`
- `assertEquals(java.lang.Object expected, java.lang.Object actual);`
- `assertEquals(java.lang.String message, double expected, double actual, double delta);`
- `assertEquals(java.lang.String message, long expected, long actual);`
- `assertEquals(java.lang.String message, java.lang.Object expected, java.lang.Object actual);`

The following deprecated Overloads are also available if needed:

- `assertEquals(double expected, double actual);`
- `assertEquals(java.lang.Object[] expected, java.lang.Object[] actual);`
- `assertEquals(java.lang.String message, java.lang.Object[] expected, java.lang.Object[] actual);`

assertTrue() and assertFalse()

The `assertTrue()` and `assertFalse()` methods can accept one or two arguments.

If two arguments are used, the first will be a message displayed in the console upon an assertion failure, and the second argument is some `boolean` to be evaluated.

The structure for `assertTrue()` and `assertFalse()` is:

```
assert<True|False>(<Message to be displayed upon failure> , <Boolean to be
evaluated>);
```

```
@Test
public void boolIsTrueTest() {
    boolean bool = true;
    assertTrue("Expected: Flag Set to True",bool);
}

@Test
public void boolIsFalseTest() {
    boolean bool = false;
    assertFalse("Expected: Flag Set to False",bool);
}
```

assertNull() and assertNotNull()

The `assertNull()` and `assertNotNull()` accept either one or two arguments.

If two arguments are supplied, the first is the message to be displayed upon assert failure, and the second being the object to be evaluated.

The structure for `assertTrue()` and `assertFalse()` is:

```
assert<Null|NotNull>(<Message to be displayed upon failure> , <Object to be
evaluated>);
```

```
@Test
public void objectIsNull() {
    Object obj = null;
    assertNull("Expected: Null Object",obj);
}

@Test
public void objectIsNotNull() {
    Object obj = "Not Null";
    assertNotNull("Expected: Not Null Object",obj);
}
```

assertSame() and assertNotSame()

The `assertSame()` & `assertNotSame()` methods both take two or three arguments, the message argument being optional.

The structure for `assertSame()` and `assertNotSame()` is:

```
assert<Same|NotSame>(<Message to be displayed upon failure> , <Object comparator
1> , <Object comparator 2>);
```

```
@Test
public void objectsAreSameTest() {
    Object obj1 = "alpha beta";
    Object obj2 = obj1;
    assertSame("Expected: Both objects are the same",obj1, obj2);
}

@Test
public void objectsAreNotSame() {
    Object obj1 = "alpha beta";
    Object obj2 = "charlie delta";
    assertNotSame("Expected: Both objects are the same",obj1, obj2);
}
```

fail()

The `fail()` method is used to outright fail a test, often used after some logical evaluation that cannot be condensed into a single assert method.

The `fail()` method takes either one argument - the message to display on test failure - or zero.

The structure for `fail()` is as follows:

```
fail(<Message to be displayed upon failure>);
```

```
@Test
public void failTest() {
    if (false) {
        fail();
    } else {
        assertTrue(true);
    }
}

@Test
public void failTestMessage() {
    if (false) {
        fail("Failed to do something");
    } else {
        assertTrue(true);
    }
}
```

Running Tests

With JUnit, we have a couple of options when running tests.

To run a single test class, simply right click the file and choose `Run As > JUnit Test`.

You can run a number of test classes at once by right clicking a parent package and choosing `Run As > JUnit Test`.

The most common versions of JUnit you'll encounter are JUnit 4 and 5. To choose which test runner to use, you can right click `Run As > Run Configurations` then change the test runner to the version you'd like.

You can also use `Coverage As > JUnit Test` to generate a coverage report for your tests. This will inform you which lines of code have been covered by your tests.

Exercises

Temperature Converter

The following code converts temperature standards:

```
public class TemperatureConverter {

    public float convertFahrenheitToCelsius(int fahrenheit) {
        return ((float) 5 / 9) * (fahrenheit - 32);
    }

    public float convertCelsiusToFahrenheit(int celsius) {
        return ((float) 9 / 5) * (celsius) + 32;
    }

    public float convertKelvinToCelsius(int kelvin) {
        return (kelvin - 273);
    }

    public float convertCelsiusToKelvin(int celsius) {
        return (celsius + 273);
    }

    public float convertKelvinToFahrenheit(int kelvin) {
        return ((float) 9 / 5) * (kelvin - 273) + 32;
    }

    public float convertFahrenheitToKelvin(int fahrenheit) {
        return ((float) 5 / 9) * (fahrenheit - 32) + 273;
    }

}
```

Write a class that tests each of these methods, using the following concepts:

- `@BeforeClass`
- `@Test`
- `assertEquals`
- `assertTrue`

Blackjack

The following code is a very rudimentary blackjack game:

```
public class BlackjackSimple {
    public static int play(int dealer, int player) {
        if (dealer > 21 && player > 21) {
            return 0;
        } else if (dealer > 21) {
            return player;
        } else if (player > 21) {
            return dealer;
        }
        return Math.max(dealer, player);
    }
}
```

Write a test suite for this code, covering as many different outcomes as you can.

Feel free to improve the blackjack code as well, but remember to test any improvements you've made!

Vending Machine

Take a look at this Git repository.

Try out the tasks in the provided `README.md`.