SOFT3202

Adam Ghanem

March 9, 2022

Contents

1 Tutorial 1 EXPORT

1.1 Welcome

1.2 Software and Prequisites

1.2.1 Gradle and Java

For this course you will need both `gradle` and `Java 17`. Use `java -version` and `gradle --version` to double check.

- Unix/WSL:
  - to get gradle and java make use of SDK if on unix/WSL
- Windows:
  - Gradle: `https://gradle.org/install/`
  - Java: `https://docs.aws.amazon.com/corretto/latest/corretto-17-ug/downloads-list.html`

Moreover, this unit will require some time and work, this is the software design pattern version of COMP2017, as you may have realised in SOFT2201.

1.2.2 JUnit

We will also be making heavy use of JUnit, which I will explain in a moment. Whether you run this through an IDE or through Gradle commands is up to you, but testing should be always at the forefront of your design.

1.2.3 Environment

Most people have been exposed to an integrated development environment (IDE). I run in a terminal/Emacs but each of you should find a balance between usability and comfort. If you are used to Eclipse and love it then try not to jump on the IntelliJ bandwagon until you have skimmed through the documentation and features.

- IntelliJ / Eclipse
- Emacs
- Vim
- terminal + code editor {vim, vscode...} + jdb

1.3 JUnit

1.3.1 Standard Example

So with JUnit, to save us having to do the long and verbose:

```
java -cp ".;./tutorials/t1/src/main/java;./tutorials/t1/src/test/java;./jar/junit.jar;
```

We can add JUnit to our gradle file and simplify the process:

```
plugins {
    id 'java'
// This is a very useful plugin to view test results.
    id 'jacoco'
}

version '1.0-SNAPSHOT'

sourceCompatibility = 1.11

repositories {
    mavenCentral()
}

tasks.withType(JavaCompile) {
    options.compilerArgs << '-Xlint:unchecked'
    options.deprecation = true
}

// This is the most important part to fetch JUnit dependecies.
dependencies {
    testImplementation group: 'junit', name: 'junit', version: '4.13.2'
    testImplementation 'org.hamcrest:hamcrest:2.2'
}

test {
    finalizedBy jacocoTestReport
}
```

What other reasons should we use Gradle for?

Well it exists as a tool that helps manage and keep track of all dependencies, this can prove to be invaluable when dependencies grow. Moreover, with versioning it keeps these dependencies in a state that works with your software.

Now since we have set up our testing environment let us test it out.

```
class SampleClass {
  public int whatsFivePlusSeven() {
    return 21;
  }
}

import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static org.junit.Assert.*;

public class TestClass {
  @Test
  public void testValidWhatsFivePlusSeven() {
    SampleClass sc = new SampleClass();
    assertEquals(12, whatsFivePlusSeven(), "You stupid.");
  }
}
```

Now we can test this by executing the first command or by running 'gradle test'.

1.3.2 Before and After

NOTE: In JUnit5 this becomes BeforeEach and AfterEach

```
public class Store {

  private List<Item> items;

  public Store() {}

  public Store(List<Item> items) {
    this.items = items;
  }

  public void addItem(Item e) {
    // What could go wrong here
    if (!contains(e) && e != null) {
      items.add(e);
    }
  }

  public boolean contains(Item e) {
    if (e.getId().equals(curr.getId())) {
      return false;
    }
    return true;
  }
}
```

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

public class TestClass {

  private Store store;


  // Say we want to load up the store with default values before every test
  @Before
  public void setupStore() {
    // We want to load the store and set it up with default values
    store = new Store();
    store.addItem(...);
    store.addItem(...);
    store.addItem(...);
    store.addItem(...);
    store.addItem(...);
    store.addItem(...);
    store.addItem(...);
    //and so on
    // but hopefully yours isn't as ugly as this.
  }

  @Test
  public void test() {
  }
}
```

Question: What is the difference between using the test case's constructor and @BeforeEach tag.

> You can still clean up exceptions thrown from @Before whereas if it comes from the constructor it will crash the test.

1.3.3 Before and After Class

Sometimes though, we may be a bit more extravagant and need to setup a connection or prepare resources ONCE before every class runs.

NOTE: We can't use a constructor because that's not how JUnitCore will run your class (constructor will run once per method). **NOTE**: This also changes to BeforeAll and AfterAll in JUnit5.

```
public class Store {
  private List<Item> items;

  public Store() {}

  public Store(List<Item> items) {
    this.items = items;
  }

  public void addItem(Item e) {
    // What could go wrong here
    if (!contains(e) && e != null) {
      items.add(e);
    }
  }

  public boolean contains(Item e) {
    if (e.getId().equals(curr.getId())) {
      return false;
    }
    return true;
  }
}


import org.junit.Test;
import org.junit.BeforeClass;
import org.junit.After;
import static org.junit.Assert.*;
import java.util.Scanner;

public class TestClass {

  private Store store;
  private Items item;

  // Now say we want to load a standard database everytime
  @BeforeAll
  public void loadItemDatabase() {
    items = new ArrayList<Item>();
    try (Scanner scan = new Scanner(new File("items.txt"))) {
      String[] itemArr = scan.nextLine().split(",");
      items.add(new Item(itemArr[0], itemArr[1]));
    } catch (FileNotFoundException e) {
      System.err.println("Error reading test file!")
        }
  }

  @Test
  public void test() {
  }

  // Now @AfterClass seems a bit redundant here, but we could in the example
  // of reading from a database of items, close the connection to the database here.
}
```

1.3.4 JUnit4 vs JUnit5

The main differences:

- when you manage dependencies in build.gradle it is a different dependency
- no longer `org.junit.Assert`, instead we use `org.junit.jupiter.Assertions.*;`
  - so basically it goes from `org.junit.whatever` to `org.junit.jupiter.whatever`
- @Before becomes @BeforeEach and same for @After
- @BeforeClass becomes @BeforeAll and same for @After

1.4 Tasks

You have a practice quiz due this week which will introduce you to the content and material.

Try running and going through some of the tutorial examples.

1.4.1 Unit Testing Examples

Given a inverse square method $\frac{1}{\sqrt{x}}$, without completing the method give me the edge-cases and unit tests you would run:

```
public class TestClass {
  public int invSqrt(int x) {
    return 0;
  }
}
```

1.4.2 Challenges

When we are testing, initially if we are doing TDD:

- there is no code to test upon.
- we do not know the implemention
- we may not even have a fully documented API
- we RED-GREEN-REFACTOR, what does this mean?

1.4.3 Simple Class

Always break down the class:

- name: SimpleClass
  - Is constructed with no arguments
  - **default constructor**
- Method: `Number getValue()`
  - Returns the current value in SimpleClass
- Method: `Boolean setValue(Number newValue)`
  - Sets the value in SimpleClass to the newValue, if it is within the range 0-100 inclusive, otherwise does nothing.
  - Returns: true if the value was set, false if it was not

We won't go straight to JUNIT we will write our tests like:

```
TEST <NAME>
<DESCRIPTION>
ASSERT <EXPECTED>

TEST getValue
Get the value of a newly created SimpleClass
Call object.getValue()

// EDGE/BOUNDARY CASE
TEST setValueEdgeCaseValid1
Set the value of the SimpleClass to 99
```

```
Call object.setValue(99)
ASSERT object.getValue() == 99

TEST setValue
Set the value of the SimpleClass to null
ASSERT object.setValue(null) == false

// Valid Cases
TEST getValueAfterValidSet
Call object.setValue(100)
ASSERT object.getValue() == 100

// Invalid Cases
TEST getValueAfterInvalidSet
ASSERT object.setValue(101) == false

TEST getValueAfterInvalidSet
CALL object.setValue(101)

// EDGE Cases with Double
TEST setValueDoubleSmall
Set the value of the SimpleClass to 1x10-9
Call object.setValue(0.0000000001)
ASSERT object.getValue() == 0.0000000001



TEST addNullOperandA
Add two valid integers
Call calculator.add(1,2)
ASSERT calculate.add(1,2) == 3

TEST setValue
Set the value for a String
Call object.setValue(new String())
ASSERT Type Error

TEST setValue
Set the value to null
Call object.setValue(null)
```

```
ASSERT false

TEST setValue
Set the value to max boundary
Call object.setValue(100)
ASSERT true
```

For the below example of SimpleClass, the first test case is not enough to check whether it is valid. Look how an issue can slip through;

```
TEST setValueInvalidPositiveEdge
  ASSERT object.setValue(101) == false

  TEST setValueInvalidPositiveEdge
  Call oldValue = object.getValue()
  ASSERT object.getValue() == oldValue
```

```
public class SimpleClass {
  private Number value;

  // What is the issue here?
  public Boolean setValue(Number number) {
    this.value = number;
    if (number.intValue() >= 0 || number.intValue() <= 100) {
      return true;
    }

    return false;
  }

  public Number getValue() {
    return this.value;
  }
}
```

1. Discussion

   No need to repeat the same type of test excessively, e.g. `add(1,2)`, `add(2,3)` etc..

   From the tutorial:

   - Is there is a lack of information in the API?

- What are the different categories of results of calling these methods you can identify?
- What boundary cases of inputs exist to achieve each of these categories?
- How independently or not independently can you test these methods?
- What key information is missing? Can we still test without it?
- Is this verification, or validation? What does the answer tell us about what we might miss if this was a real project?
- We can instantiate number as:
  - `Number d = new Double(0.00000001)`
  - `Number i = new Integer(0)`

1.4.4 Shopping Basket

Let us breakdown the ShoppingBasket:

- are there any constructors and what is the constructor?
- what methods do we have and what parameters do they take?
- List<Pair<String, Number»

We are not programming or building this class we are just testing it. Below are just examples we found during class:

```
TEST testConstructorNoExceptions
  Call ShoppingBasket sb = new ShoppingBasket();


// TEST addItemDifferentItem
// Add item that doesn't exist, a pineapple.
// Call ShoppingBasket sb = new ShoppingBasket();
// Call sb.addItem("pineapple", 1)
// ASSERT IllegalArgumentException thrown

public class ShoppingBasketTest {

  @Test
  public void addItemDifferentItem() {
    ShoppingBasket sb = new ShoppingBasket();
    boolean thrown = false;
    try {
      sb.addItem("pineapple", 1);
    } catch (IllegalArgumentException e) {
      thrown = false;
    }
    assertTrue(thrown, "Should have thrown an IllegalArgumentException");
  }


  // TEST addItemNegativeCount
  // Call addItem with a valid item but negative count.
  // CALL sb.addItem("apple", 0);
  // ASSERT NumberException thrown
  @Test(expected=NumberException.class)
  public void addItemNegativeCount() {

    ShoppingBasket sb = new ShoppingBasket();
    boolean thrown = false;
    sb.addItem("apple", 0);
  }
}


TEST  addItemNullItem
Add a null item but non-null count to addItem
Call object.addItem(null, 10)
ASSERT IllegalArgumentException

TEST addItemNullCount
Add a null count but non-null item to addItem
Call object.addItem("persimmon", null)
ASSERT NumberException || IllegalArgumentException

TEST addItemNullItemAndCount
Call object.addItem(null, null)
ASSERT IllegalArgument

TEST addItemZeroCountInvalidItem
```

```
Call object.addItem("asparagus", 0)
ASSERT IllegalArgument

TEST addItemZeroCountValidItem
Call object.addItem("apple", 0)
ASSERT NUmberException

TEST addItemListNotNull
Call sb = new ShoppingBasket()
ASSERT
```