INFO1113 Object-Oriented Programming

Week 8B: Testing

Copyright Warning

Topics

- Assert Keyword (s. 4)
- JUnit (s. 25)
- JUnit API and Testing Code (s. 42)

Assert keyword

Java includes support for the **assert** keyword that allows checking for the truthiness of an expression.

The assert keyword is used in conjunction with a boolean expression.

Refer to Programming with assertions, Preconditions, Postconditions, and Class Invariants, (https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html#usage-conditions)

Assert keyword

Assert evaluates an expression and will throw an **AssertionError** if the statement if false.

Syntax:

```
assert expression
```

Assert keyword

Assert evaluates an expression and will throw an **AssertionError** if the statement if false.

Syntax:

```
assert expression
```

Example:

```
assert list.size() > 0

assert list.size() == 0 && writtenFiles
```

What happens if the condition is false?

Assert keyword

Assert evaluates an expression and will throw an **AssertionError** if the statement if false.

Syntax:

```
assert expression
```

As discussed previous, since it throws an **Error** type, it will cause our application to crash.

Assert keyword

assert is is a keyword that allows us to test the truthiness of a method or variable.

You would utilise this feature in an attempt to ensure that your program is sound. We are able to test preconditions, postconditions and anything in between.

However, assert is not a substitute for control flow. The feature is to highlight anything you deem incorrect in your application.

Refer to Programming with assertions, Preconditions, Postconditions, and Class Invariants, (https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html#usage-conditions)

Post-condition

A post-condition is where any mutation or output from a method is considered to adhere to the requirements of the method.

Simply: What the method promises to do.

For example, A method must return the sum of numbers in a list. Failing this results in the post-condition being false.

Refer to Programming with assertions, Preconditions, Postconditions, and Class Invariants, (https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html#usage-conditions)

Where would we want to use it?

Assert keyword

Any place where we want to cause a failure because a condition within the program is not met.

However, it can be difficult to consider why we may want this, considering most states within our program can be be recoverable.

Assert keyword

A few scenarios where it is applicable:

- Preparing to write updates to an operating system or large block of software
- Failure to write a core file that is necessary to your application running correctly.
- Checking that methods provide the correct result and modifications to objects.

So how would it be formed?

Assert keyword

Any place where we want to cause a failure because a condition within the program is not met.

However, it can be difficult to consider why we may want this, considering most states within our program can be be recoverable.

Assert keyword

Let’s take a look at the following program

```
import java.util.List;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

public class PackageInstaller {

   //<snipped>

    private void preCheck() {
        File f = new File(pathPrefix);
        assert f.exists();
        assert f.isDirectory();
        assert key != null;
        assert keyInput != null;
        assert key.verify(keyInput);
        assert noFiles > 0;
        assert files != null;
        assert files.size() > 0;
        assert files.size() == noFiles;
        assert noFilesWritten == 0;
    }

}
```

```
//<rest continues here>
private void commit() {

        for(File file : files) {
            try {
            Files.copy(file.toPath(),
                (new File(pathPrefix + file.getName())).toPath(),
                StandardCopyOption.REPLACE_EXISTING);
                noFilesWritten++;
            } catch(IOException e) {}
        }
    }

    public void install() {
        preCheck();
        commit();
        postCheck();
        cleanup();
    }

    private void postCheck() {
        assert noFilesWritten > 0;
        assert noFilesWritten == files.size();
        for(File file : files) {
            assert new File(pathPrefix+file.getName()).exists();
        }
    }
```

Assert keyword

Let's take a look at the following program

```
import java.util.List;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

public class PackageInstaller {

   //<snipped>

    private void preCheck() {
        File f = new File(pathPrefix);
        assert f.exists();
        assert f.isDirectory();
        assert key != null;
        assert keyInput != null;
        assert key.verify(keyInput);
        assert noFiles > 0;
        assert files != null;
        assert files.size() > 0;
        assert files.size() == noFiles;
        assert noFilesWritten == 0;
    }

}
```

```
//<rest continues here>
private void commit() {

        for(File file : files) {
            try {
            Files.copy(file.toPath(),
                (new File(pathPrefix + file.getName())).toPath(),
                StandardCopyOption.REPLACE_EXISTING);
                noFilesWritten++;
            } catch(IOException e) {}
        }
    }

    public void install() {
        preCheck();
        commit();
        postCheck();
        cleanup();
    }

    private void postCheck() {
        assert noFilesWritten > 0;
        assert noFilesWritten == files.size();
                                  iles) {
                                  le(pathPrefix+file.getName()).exists();
```

The main method that will be called by our installer object is the **install()** method. This has simple list of instructions to carry out.

Assert keyword

Let's take a look at the following program

```
import java.util.List;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

public class PackageInstaller {

   //<snipped>

   private void preCheck() {
        File f = new File(pathPrefix);
        assert f.exists();
        assert f.isDirectory();
        assert key != null;
        assert keyInput != null;
        assert key.verify(keyInput);
        assert noFiles > 0;
        assert files != null;
        assert files.size() > 0;
        assert files.size() == noFiles;
        assert noFilesWritten == 0;
   }

}
```

```
//<rest continues here>
private void commit() {

        for(File file : files) {
            try {
            Files.copy(file.toPath(),
                (new File(pathPrefix + file.getName())).toPath(),
                StandardCopyOption.REPLACE_EXISTING);
                noFilesWritten++;
            } catch(IOException e) {}
        }
    }

    public void install() {
        preCheck();
        commit();
        postCheck();
        cleanup();
    }

    private void postCheck() {
        assert noFilesWritten > 0;
        assert noFilesWritten == files.size();
        for(File file : files) {
            assert new File(pathPrefix+file.getName()).exists();
```

The first method being the **preCheck()** method that will need to verify if all dependencies for the installation are satisfied.

Assert keyword

Let's take a look at the following program

```
import java.util.List;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

public class PackageInstaller {

   //<snipped>

    private void preCheck() {
        File f = new File(pathPrefix);
        assert f.exists();
        assert f.isDirectory();
        assert key != null;
        assert keyInput != null;
        assert key.verify(keyInput);
        assert noFiles > 0;
        assert files != null;
        assert files.size() > 0;
        assert files.size() == noFiles;
        assert noFilesWritten == 0;
    }

}
```

```
//<rest continues here>
private void commit() {

        for(File file : files) {
            try {
            Files.copy(file.toPath(),
                (new File(pathPrefix + file.getName())).toPath(),
                StandardCopyOption.REPLACE_EXISTING);
                noFilesWritten++;
            } catch(IOException e) {}
        }
    }

    public void install() {
        preCheck();
        commit();
        postCheck();
        cleanup();
    }

    private void postCheck() {
        assert noFilesWritten > 0;
        assert noFilesWritten == files.size();
        for(File file : files) {
            ...le(pathPrefix+file.getName()).exists();
```

Each assert potentially will prevent the installer from progressing if it fails the check

Assert keyword

Let's take a look at the following program

```
import java.util.List;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

public class PackageInstaller {

   //<snipped>

    private void preCheck() {
        File f = new File(pathPrefix);
        assert f.exists();
        assert f.isDirectory();
        assert key !=
        assert keyInp
        assert key.ve
        assert noFile
        assert files != null;
        assert files.size() > 0;
        assert files.size() == noFiles;
        assert noFilesWritten == 0;
    }

}
```

If a precheck passes, then move to writing the files to the directory specified.

```
//<rest continues here>
private void commit() {

        for(File file : files) {
            try {
            Files.copy(file.toPath(),
                (new File(pathPrefix + file.getName())).toPath(),
                StandardCopyOption.REPLACE_EXISTING);
                noFilesWritten++;
            } catch(IOException e) {}
        }
    }

    public void install() {
        preCheck();
        commit();
        postCheck();
        cleanup();
    }

    private void postCheck() {
        assert noFilesWritten > 0;
        assert noFilesWritten == files.size();
        for(File file : files) {
            assert new File(pathPrefix+file.getName()).exists();
        }
    }
```

What potential problems could have happened if we didn’t check prior to writing?

Assert keyword

Let's take a look at the following program

```
import java.util.List;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

public class PackageInstaller {

   //<snipped>

    private void preCheck() {
        File f = new File(pathPrefix);
        assert f.exists();
        assert f.isDirectory();
        assert key != null;
        assert keyInput != null;
        assert key.verify(keyInput);
        assert noFiles > 0;
        assert files != null;
        assert files.size() > 0;
        assert files.size() == noFiles;
        assert noFilesWritten == 0;
    }

}
```

```
//<rest continues here>
private void commit() {

        for(File file : files) {
            try {
            Files.copy(file.toPath(),
                (new File(pathPrefix + file.getName())).toPath(),
                StandardCopyOption.REPLACE_EXISTING);
                noFilesWritten++;
            } catch(IOException e) {}
        }
    }

    public void install() {
        preCheck();
        commit();
        postCheck();
        cleanup();
    }

    private void postCheck() {
        assert noFilesWritten > 0;
        assert noFilesWritten == files.size();
        for(File file : files) {
            assert new File(pathPrefix+file.getName()).exists();
        }
    }
```

We will run checks after writing the files to ensure that they have been written.

Why would we need to check after writing?

Assert keyword

Let's take a look at the following program

```
import java.util.List;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

public class PackageInstaller {

   //<snipped>

    private void preCheck() {
        File f = new File(pathPrefix);
        assert f.exists();
        assert f.isDirectory();
        assert key != null;
        assert keyInput != null;
        assert key.verify(keyInput);
        assert noFiles > 0;
        assert files != null;
        assert files.size() > 0;
        assert files.size() == noFiles;
        assert noFilesWritten == 0;
    }
}
```

We need to check that all files were written as the commit method can skip files if an exception occurs. **What other strategies could we employ?**

```
//<rest continues here>
private void commit() {

        for(File file : files) {
            try {
            Files.copy(file.toPath(),
                (new File(pathPrefix + file.getName())).toPath(),
                StandardCopyOption.REPLACE_EXISTING);
                noFilesWritten++;
            } catch(IOException e) {}
        }
    }

    public void install() {
        preCheck();
        commit();
        postCheck();
        cleanup();
    }

    private void postCheck() {
        assert noFilesWritten > 0;
        assert noFilesWritten == files.size();
        for(File file : files) {
            assert new File(pathPrefix+file.getName()).exists();
        }
    }
```

JUnit

Although the compiler performs quite a number of checks for us to ensure we are using types correctly, it doesn't ensure that our program logic is infallible.

When building any meaningful software project you will need to formulate a mechanism of testing the software complies with the requirements.

For more information about JUnit outside of this lecture, visit: https://junit.org/junit5/docs/current/user-guide/

JUnit

A common testing framework in the Java ecosystem is **JUnit**. You have written your own test classes to check if your code is performing correctly.

JUnit gives us a simple framework that allows us to mark methods as tests.

White Box Testing - This is typically where we employ some unit testing software, to help analyse the internals of the system and test them independently.

Black Box Testing - User centric testing, without knowledge of the internals, input is given and compared to match the output of the program.

Regression Testing, When the system has been modified and the changes may result in a failure of a previous successful test case.

Integration Testing, When developing individual components, we want to integrate it into the whole system and check to see if it works.

JUnit

To set up JUnit you need to acquire **junit.jar** and **hamcrest.jar** files that are used to run **JUnit**.

Within the java ecosystem **.jar** files (Java Archive) are a collection of **.class** files that we can import into our own application. It exposes a whole new set of methods.

JUnit

The Java platform is typically geared towards IDEs and sometimes it can be troublesome to strictly live in a command line world with Java.

This isn’t true for a lot of environments which have a different set of tooling that affords interaction with a command line interface and an IDE.

Within **JUnit** we have access to variety of **annotations** that allow us to determine an order of execution for some of our methods and also sort test execution if so needed.

However, we should not need to **order** test cases but we may need to create **a preparation** method.

The annotations where we can use with methods.

`@Test`

Simply, this annotates a method as a test method and will be considered part of the results.

`@Before`

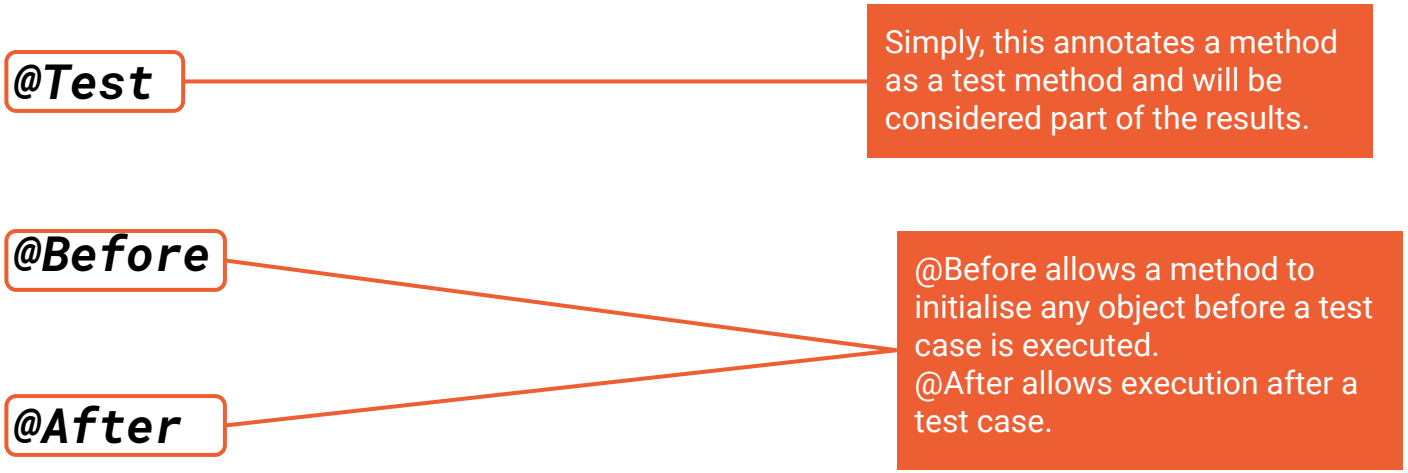
`@After`

`@BeforeClass`

`@AfterClass`

Refer to JUnit API Documentation, (https://junit.org/junit5/docs/current/api/overview-summary.html)

The annotations where we can use with methods.

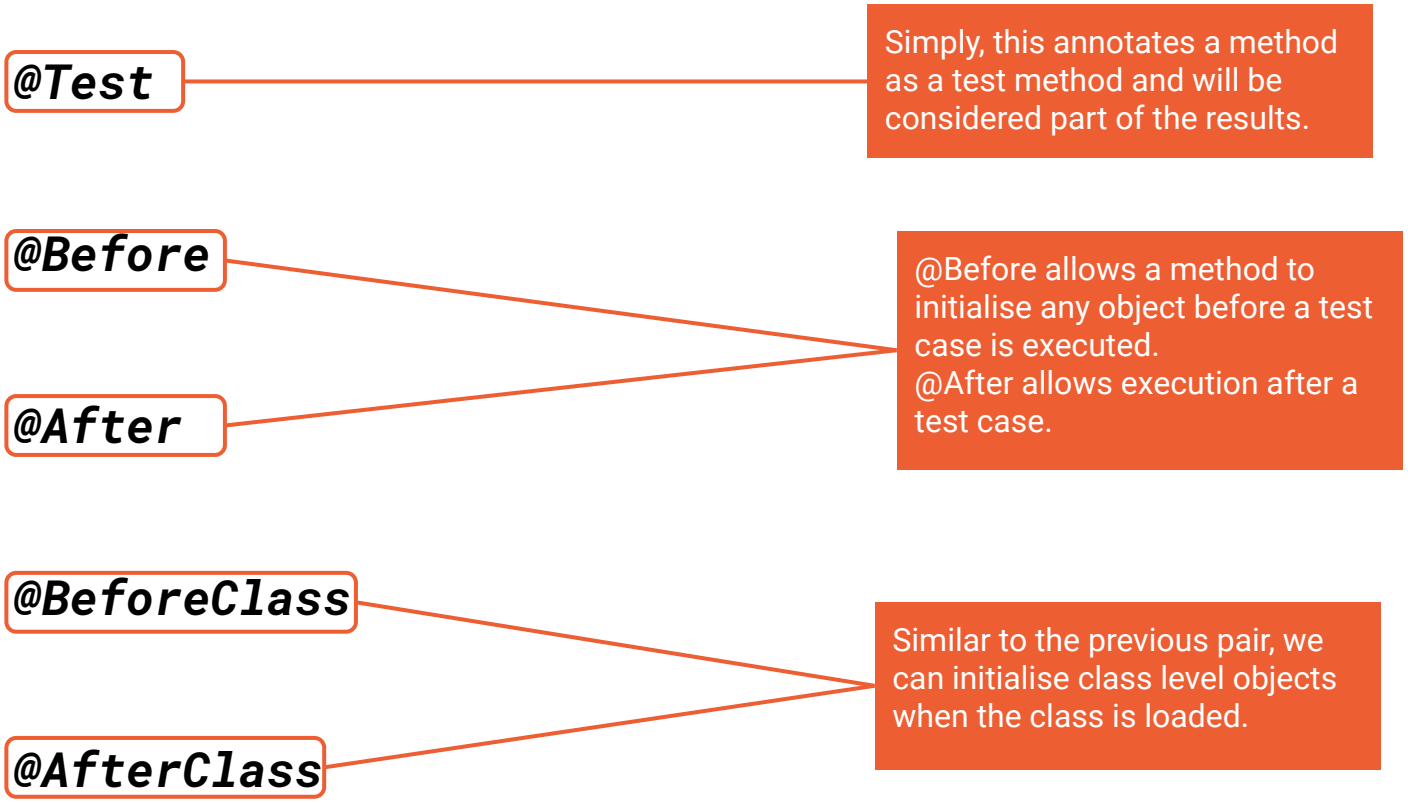


@BeforeClass

@AfterClass

Refer to JUnit API Documentation, (https://junit.org/junit5/docs/current/api/overview-summary.html)

The annotations where we can use with methods.



Refer to JUnit API Documentation, (https://junit.org/junit5/docs/current/api/overview-summary.html)

Testing a for a simple null

```
@Test
public void checkForNull() {
    Container a = new Container(null);
    assertNull(a.get());
}
```

Testing a for a simple null

@Test, provides annotation of the method that it is a test case.

```
@Test
public void checkForNull() {
    Container a = new Container(null);
    assertNull(a.get());
}
```

Testing a for a simple null

```
@Test
public void checkForNull() {
    Container a = new Container(null);
    assertNull(a.get());
}
```

@Test, provides annotation of the method that it is a test case.

We can use the JUnit library methods to test if it is true.

Our assert methods we have available within our JUnit.

```
assertTrue( boolean expression )

assertFalse( boolean expression )

assertEquals( expected , actual )

assertNull( object )

assertSame( object1 , object2 )
```

Our assert methods we have available within our JUnit.

```
assertTrue( boolean expression )
```

```
assertFalse( boolean expression )
```

They accept boolean expressions that should hold true or false (depending on what you expect the result to be)

```
assertEquals( expected , actual )
```

```
assertNull( object )
```

```
assertSame( object1 , object2 )
```

Our assert methods we have available within our JUnit.

```
assertTrue( boolean expression )
```

```
assertFalse( boolean expression )
```

They accept boolean expressions that should hold true or false (depending on what you expect the result to be)

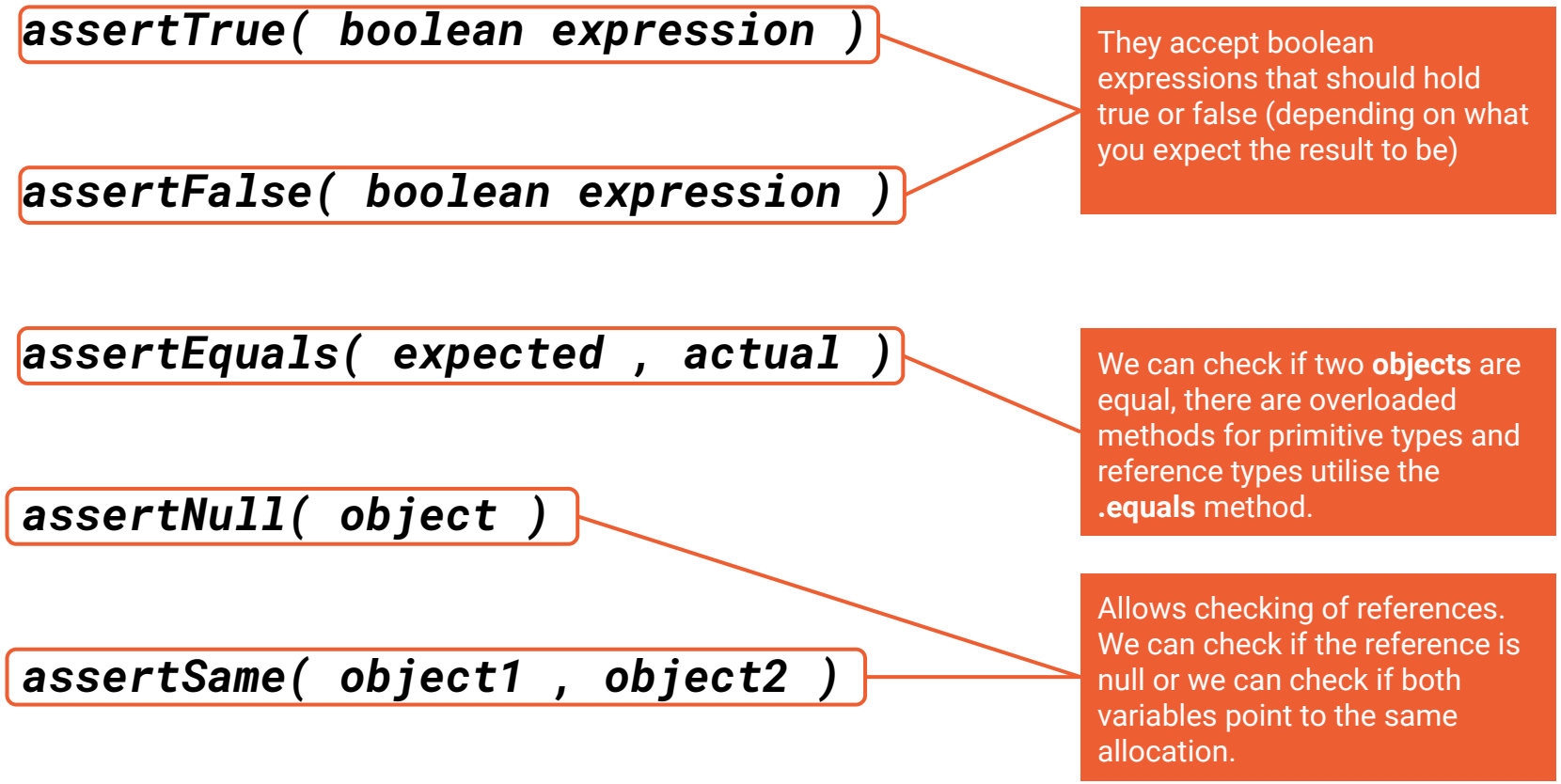
```
assertEquals( expected , actual )
```

We can check if two **objects** are equal, there are overloaded methods for primitive types and reference types utilise the **.equals** method.

```
assertNull( object )
```

```
assertSame( object1 , object2 )
```

Our assert methods we have available within our JUnit.



Let’s write a simple test file for a calculator

JUnit

Once we have constructed our test case, we will need to compile it with the junit and hamcrest archives.

```
> javac -cp .:junit-4.12.jar:hamcrest-core-1.3.jar MyTestClass.java
```

JUnit

Once we have constructed our test case, we will need to compile it with the junit and hamcrest archives.

```
> javac -cp .:junit-4.12.jar:hamcrest-core-1.3.jar MyTestClass.java
```

This is a **classpath** flag that allows us to specify the location of other classes that we can use during compilation

Once we have constructed our test case, we will need to compile it with the junit and hamcrest archives.

```
> javac -cp .:junit-4.12.jar:hamcrest-core-1.3.jar MyTestClass.java
```

This is a **classpath** flag that allows us to specify the location of other classes that we can use during compilation

We specify our java archives within this section, separated with :

JUnit

To execute a JUnit class, we need to run the program differently from before.

```
> java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore MyTestClass
```

Because we are utilising classes in a java archive file, we will need to refer to that when executing our program.

JUnit

To execute a JUnit class, we need to run the program differently from before.

```
> java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore MyTestClass
```

Because we are utilising classes in a java archive file, we will need to refer to that when executing our program.

We will utilise the class **JUnitCore** which will run our class.

JUnit

To execute a JUnit class, we need to run the program differently from before.

```
> java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore MyTestClass
```

```
> java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore MyTestClass

JUnit version 4.12
..
Time: 0.003

OK (2 tests)
```

Let’s test our LinkedList from last week

See you next time!