Level 3

Test Runners Frameworks

3.1 - Introduction to TestNG/JUnit

TestNG and **JUnit** are testing frameworks that allow us to manage and configure our test cases as we want.

Some of the features that TestNG and jUnit provide us are:

- Annotations
- Parallel running
- Suite creator
- Setting up of parameters
- Data-driven testing
- Support for Eclipse, Maven, and other IDEs
- Listeners

To be able to use TestNG in our project we need to add the dependency in Maven’s **pom.xml** as we did in the previous lesson.

In addition to this, if we want an integration with Eclipse, we must install the TestNG plugin in Eclipse as follows:

http://testna.org/doc/eclipse.html

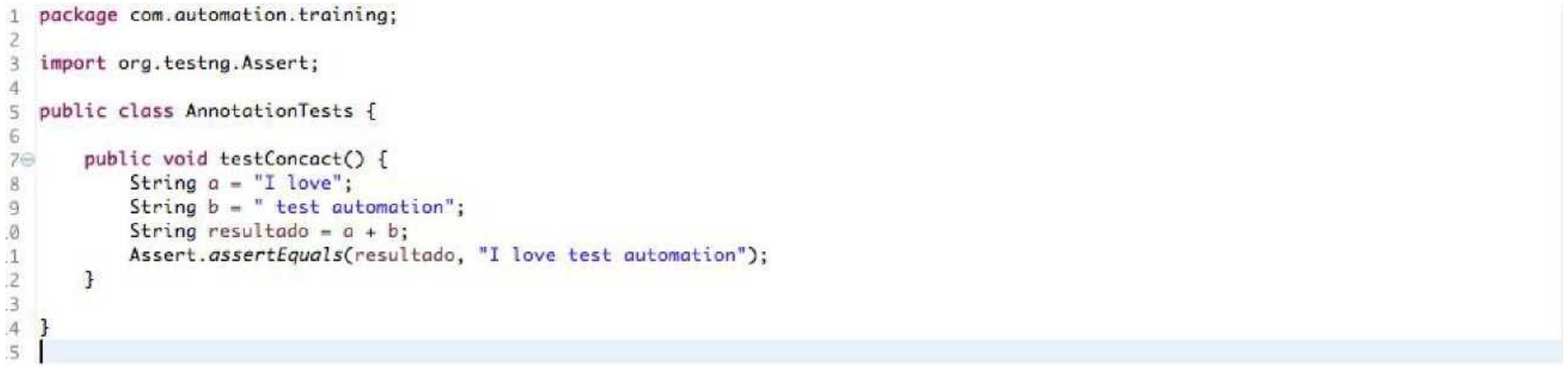
3.2. - Annotation

An annotation is a tag that allows the indication of different things; it can be applied to a method, a class, a variable or a parameter.

TestNG uses annotations to allow us to manage and indicate what we are doing. For example, passing parameters and configurations to our test cases and methods, saying that a method is a test, indicating that a method is a data provider.

Let's see how they are used and some examples:

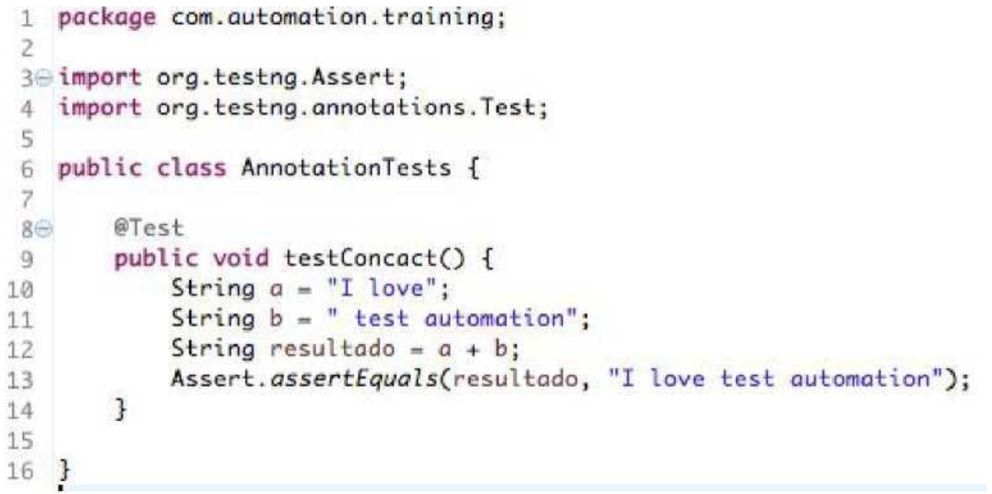
Within **com.automation.training** we create a class called **AnnotationTests** and then create a method that validates a function of the String class:



```java
package com.automation.training;

import org.testng.Assert;

public class AnnotationTests {

    public void testConcact() {
        String a = "I love";
        String b = " test automation";
        String resultado = a + b;
        Assert.assertEquals(resultado, "I love test automation");
    }

}

```

As you can see, it is a fairly simple method that creates two strings, concatenates them and finally validates that the result of the concatenation is what we expect.

To run this method, we should create from the Main class an instance of the class and call it from there, but TestNG takes care of this automatically, only adding the annotation @Test on the method:

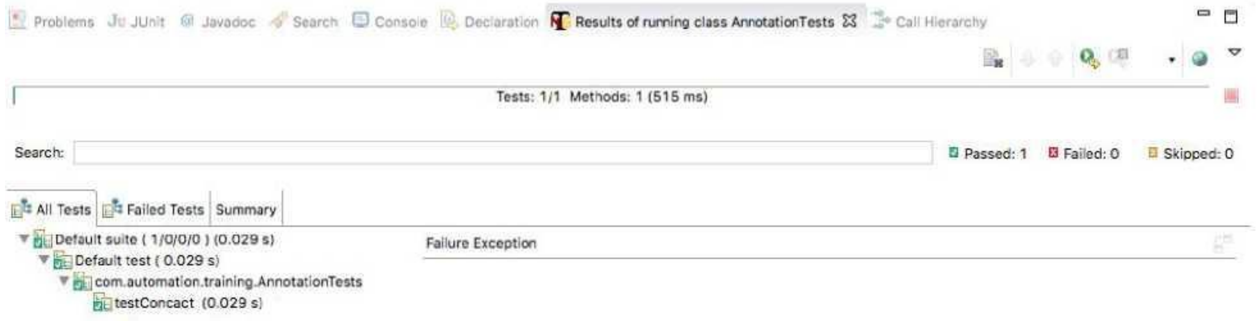


```java
package com.automation.training;

import org.testng.Assert;
import org.testng.annotations.Test;

public class AnnotationTests {

    @Test
    public void testConcact() {
        String a = "I love";
        String b = " test automation";
        String resultado = a + b;
        Assert.assertEquals(resultado, "I love test automation");
    }

}
```

Now we are going to run it in Eclipse; in order to do so, we simply need to right-click on the class and select:

Run as -> TestNG test

The test case will run and we will see the result at the bottom of the screen:



```
Tests: 1/1  Methods: 1 (515 ms)

Passed: 1   Failed: 0   Skipped: 0

Default suite ( 1/0/0/0 ) (0.029 s)
  Default test ( 0.029 s)
    com.automation.training.AnnotationTests
      testConcact  (0.029 s)

Failure Exception
```

3.2.2 - How to use annotations

The annotations allow us to configure our test cases, setting up parameters for them, order them, etc.

Let's see some of the most important annotations in testNG:

@BeforeSuite: The annotated method will only run once before the suite, that is,

before all the test cases

@AfterSuite: The annotated method will only run once before the suite,

that is, before all the test cases

@BeforeClass: The annotated method will only run once before all the tests of the class where it is found

@AfterClass: The annotated method will only run once after

all the tests of the class where it is found

@BeforeTest: The annotated method will only run once before each test

of the classes included in the tag <test>

@AfterTest: The annotated method will only run once after each test of the

classes included in the tag <test>

@BeforeMethod: The annotated method will only run once before

each test method

@AfterMethod: The annotated method will only run once after each test method

@DataProvider: The annotated method will be a data provider, which we will then use in the tests to obtain parameters. It must be a method that returns an **Object[][]**.

@Listeners: Defines a listener in a test class

@Test: Defines a method or class as part of a test

3.3 - Before and After. How to use them in our cases

This annotation helps us when we want a certain action to be carried out before each test case. As an example, let’s suppose we have test cases that require logging in with a certain user before running, and need to log off at the end of the execution.

Instead of having N test cases that do the following

Test1:

Login

Steps

Logout

Test2:

Login

Steps

Logout

Test3:

Login

Steps

Logout

We can define a BeforeMethod that performs the login, and an AfterMethod that performs a logout. Then our tests will only perform the steps:

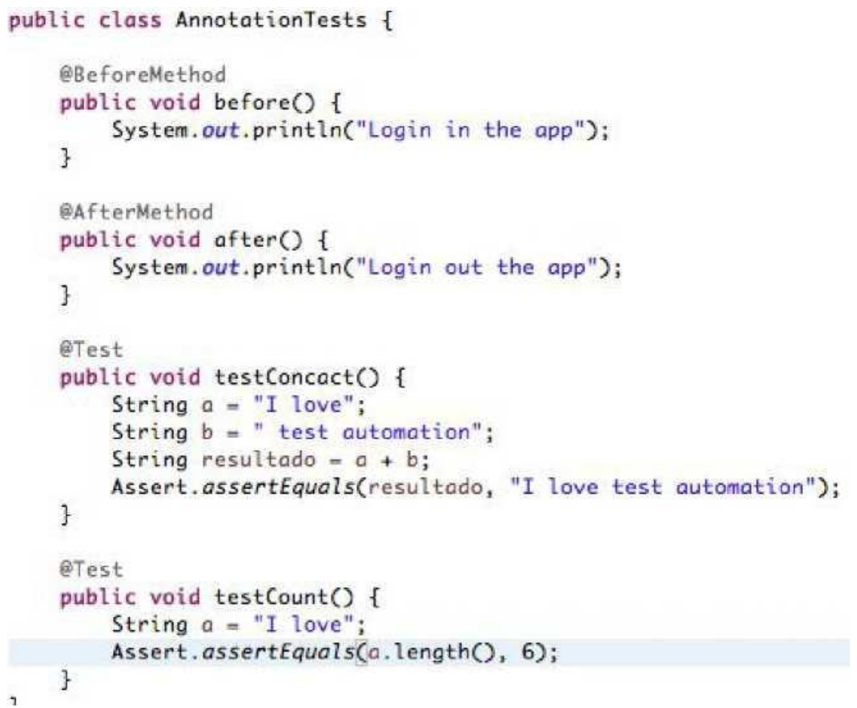
BeforeMethod:
Login

AfterMethod:
Logout

Test1:
Steps

Test2:
Steps

Test3:
Steps



```java
public class AnnotationTests {

    @BeforeMethod
    public void before() {
        System.out.println("Login in the app");
    }

    @AfterMethod
    public void after() {
        System.out.println("Login out the app");
    }

    @Test
    public void testConcact() {
        String a = "I love";
        String b = " test automation";
        String resultado = a + b;
        Assert.assertEquals(resultado, "I love test automation");
    }

    @Test
    public void testCount() {
        String a = "I love";
        Assert.assertEquals(a.length(), 6);
    }
}
```

If we see the console, we will see that it was printed:

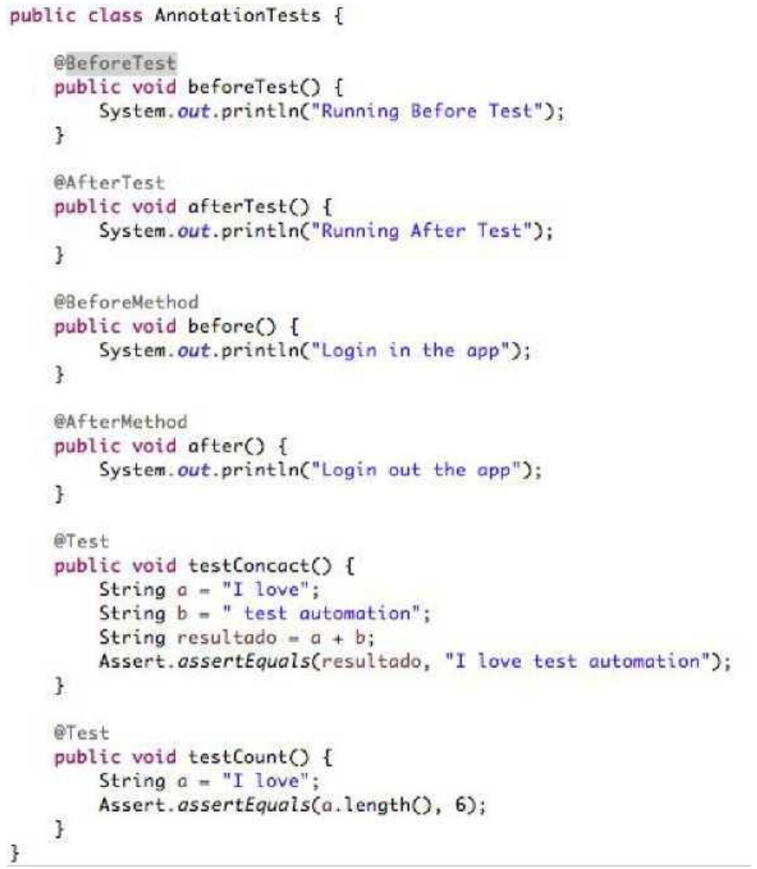
- Login in the app
- Login out the app
- Login in the app
- Login out the app

That is to say, before each test, the before method was run and upon completion, the after method was run.

It should be noted that this is an example of little use in real life, but it is useful to see the behavior of this annotation.

3.3.2 - Before and After test.

In this case, we will see how this annotation behaves, which will run the annotated method before the complete test, and after the complete test. As a clarification, a test in this case does not refer to the test case alone, but to the set of test cases that do the test. Let's see how it behaves:



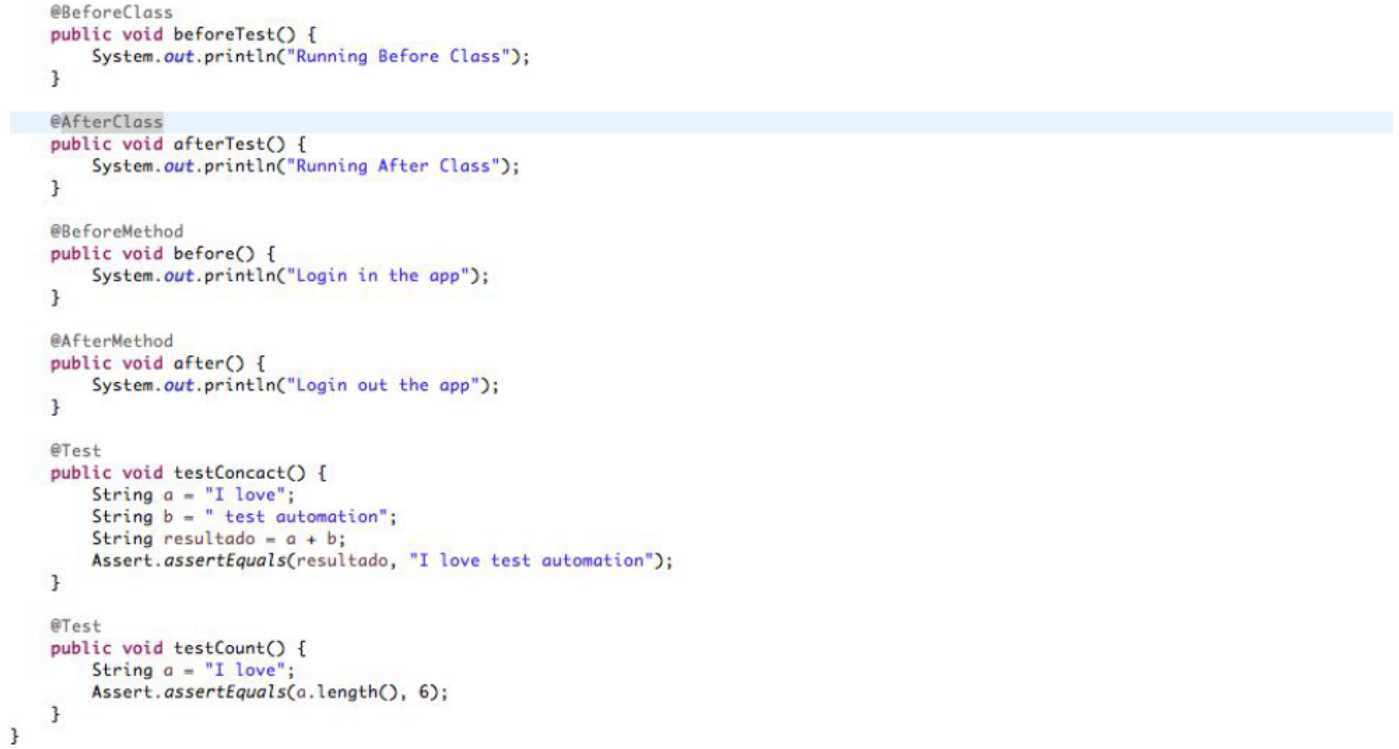
```java
public class AnnotationTests {

    @BeforeTest
    public void beforeTest() {
        System.out.println("Running Before Test");
    }

    @AfterTest
    public void afterTest() {
        System.out.println("Running After Test");
    }

    @BeforeMethod
    public void before() {
        System.out.println("Login in the app");
    }

    @AfterMethod
    public void after() {
        System.out.println("Login out the app");
    }

    @Test
    public void testConcact() {
        String a = "I love";
        String b = " test automation";
        String resultado = a + b;
        Assert.assertEquals(resultado, "I love test automation");
    }

    @Test
    public void testCount() {
        String a = "I love";
        Assert.assertEquals(a.length(), 6);
    }
}
```

When executing this code we will see the following output in the console:

- Running Before Test
- Login in the app
- Login out the app
- Login in the app

- Login out the app
- Running After Test
- PASSED: testConcact
- PASSED: testCount

These annotations will run before and after each class included in our tests. As an example, let's see the following code:



```
    @BeforeClass
    public void beforeTest() {
        System.out.println("Running Before Class");
    }

    @AfterClass
    public void afterTest() {
        System.out.println("Running After Class");
    }

    @BeforeMethod
    public void before() {
        System.out.println("Login in the app");
    }

    @AfterMethod
    public void after() {
        System.out.println("Login out the app");
    }

    @Test
    public void testConcact() {
        String a = "I love";
        String b = " test automation";
        String resultado = a + b;
        Assert.assertEquals(resultado, "I love test automation");
    }

    @Test
    public void testCount() {
        String a = "I love";
        Assert.assertEquals(a.length(), 6);
    }
}
```

This returns the following console output

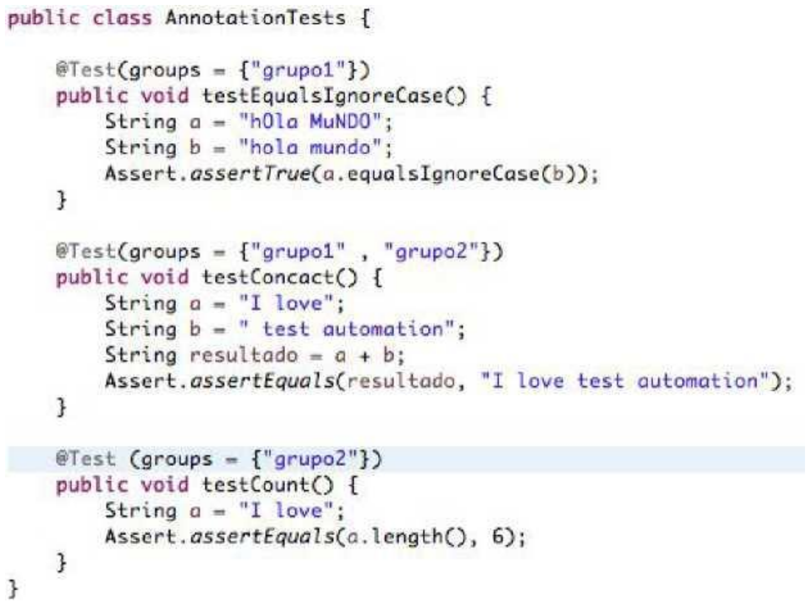
- Running Before Class
- Login in the app
- Login out the app
- Login in the app
- Login out the app
- Running After Class
- PASSED: testConcact
- PASSED: testCount

3.4 - Creating groups

Many times we have several test cases that we want to tag somehow to be able to select them when making executions. As an example, we can say that in a simple web app, we will have tests dedicated to testing the login, others dedicated to testing the user record, and others dedicated to testing the shopping cart of the application.

In this case, it is useful that each of the tests is tagged with a label that allows, for example, to run each test belonging to the login, quickly.

In our example we are going to add a third test and create two groups:



```java
public class AnnotationTests {

    @Test(groups = {"grupo1"})
    public void testEqualsIgnoreCase() {
        String a = "hOla MuNDO";
        String b = "hola mundo";
        Assert.assertTrue(a.equalsIgnoreCase(b));
    }

    @Test(groups = {"grupo1" , "grupo2"})
    public void testConcact() {
        String a = "I love";
        String b = " test automation";
        String resultado = a + b;
        Assert.assertEquals(resultado, "I love test automation");
    }

    @Test (groups = {"grupo2"})
    public void testCount() {
        String a = "I love";
        Assert.assertEquals(a.length(), 6);
    }
}
```

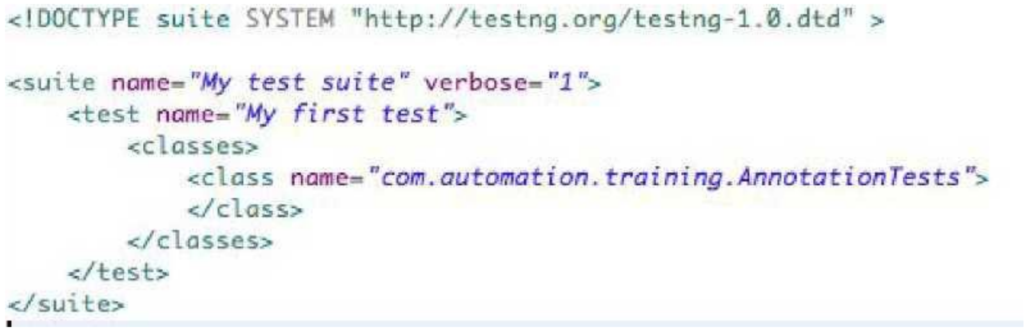
In this example we see how the first case belongs to a group called “groupl", the second case belongs simultaneously to “groupl" and "group2" and the third belongs only to "group2".

Then, from our suite, we can specify which group we want to run, or which group we want to exclude. We'll see this in the next topic, "Creating Suites."

3.5 - Creating suites

A suite can be described as a set of configurations that tell testNg how to run our test cases. In it we can configure parameters to say, for example, which browser we want to use, or in what environment we want to run something, or select the groups we want to run or exclude, as well as indicate the listeners we want to include.

It is an xml file, let's see a basic one:

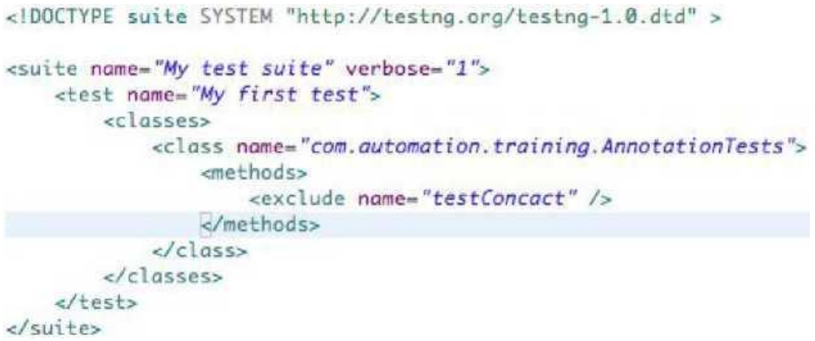


```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="My test suite" verbose="1">
    <test name="My first test">
        <classes>
            <class name="com.automation.training.AnnotationTests">
            </class>
        </classes>
    </test>
</suite>
```

This suite indicates that we want to run all the methods from the training class. In turn, it indicates that our test is called "My first test" and that the suite is called "My test suite.”

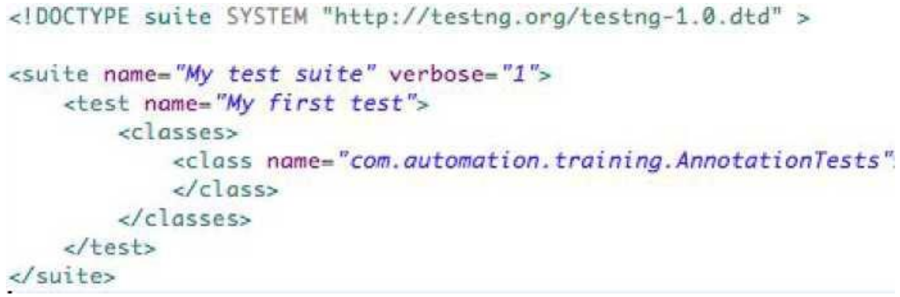
Excluding methods

Let's see a case where we want to exclude a certain method from our execution:



```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="My test suite" verbose="1">
    <test name="My first test">
        <classes>
            <class name="com.automation.training.AnnotationTests">
                <methods>
                    <exclude name="testConcact" />
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

A suite can be described as a set of configurations that tell testNg how to run our test cases. In it we can configure parameters to say, for example, which browser we want to use, or in what environment we want to run something, or select the groups we want to run or exclude, as well as indicate the listeners we want to include.

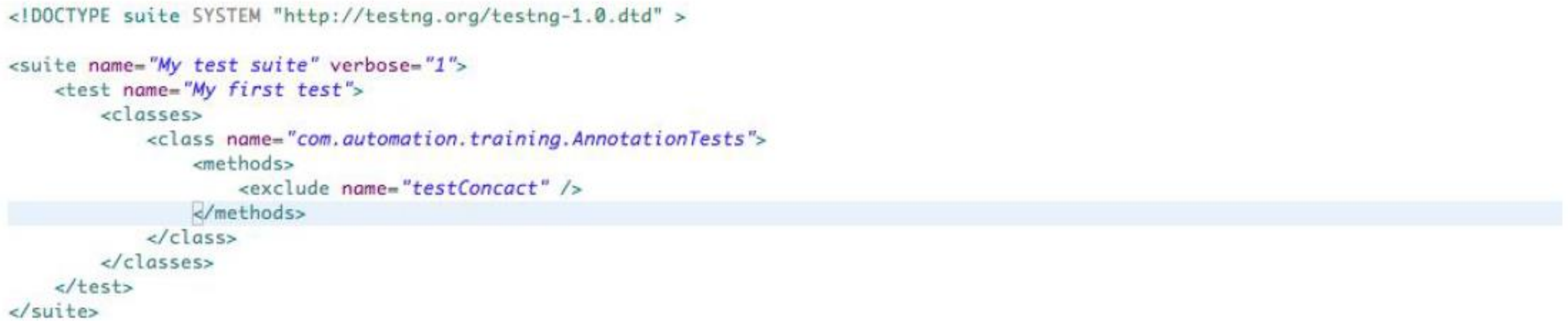


```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="My test suite" verbose="1">
    <test name="My first test">
        <classes>
            <class name="com.automation.training.AnnotationTests"
            </class>
        </classes>
    </test>
</suite>
```

This suite indicates that we want to run all the methods from the training class. In turn, it indicates that our test is called "My first test" and that the suite is called "My test.”

Excluding methods:

Let's see a case where we want to exclude a certain method from our execution:

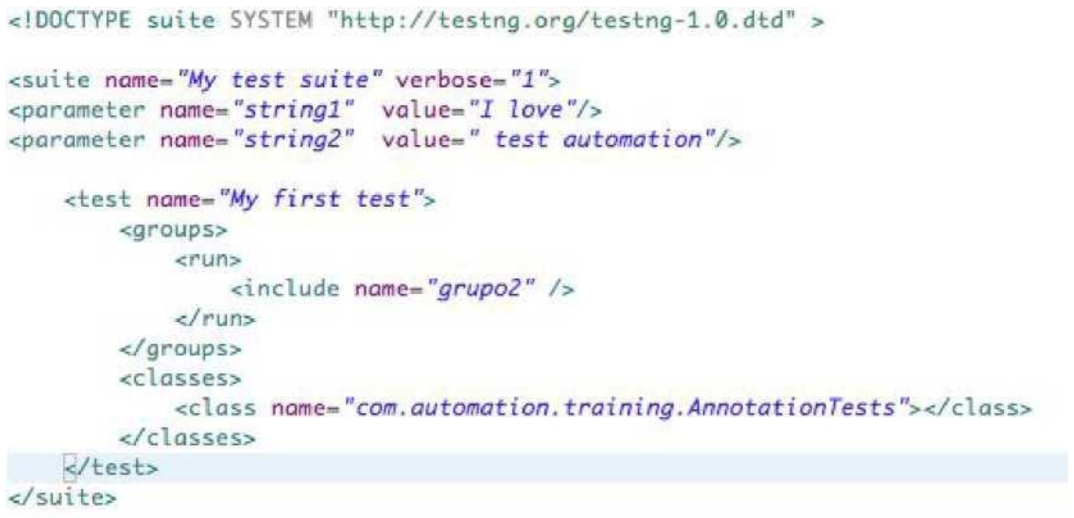


```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="My test suite" verbose="1">
    <test name="My first test">
        <classes>
            <class name="com.automation.training.AnnotationTests">
                <methods>
                    <exclude name="testConcact" />
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

3.5.2 - Using parameters

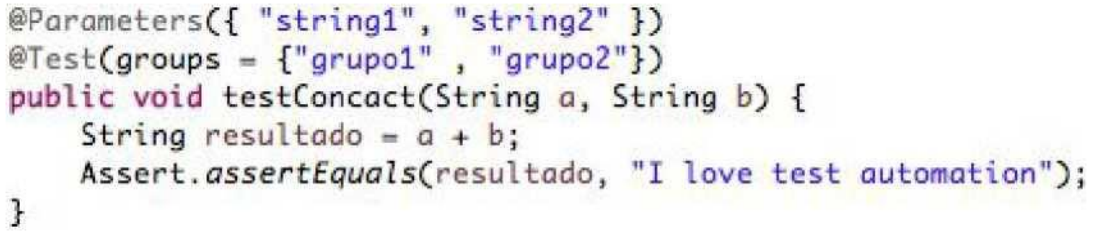
TestNG allows the configuration of different parameters in our suite to be used in tests. They are simply created in the xml configuration file and read from our tests when we need them.

To create a parameter in the suite we must do the following:



```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="My test suite" verbose="1">
<parameter name="string1"  value="I love"/>
<parameter name="string2"  value=" test automation"/>

    <test name="My first test">
        <groups>
            <run>
                <include name="grupo2" />
            </run>
        </groups>
        <classes>
            <class name="com.automation.training.AnnotationTests"></class>
        </classes>
    </test>
</suite>
```

In this example we create two parameters called string1 and string2 respectively. Now we are going to set parameters for one of the previously written test cases:

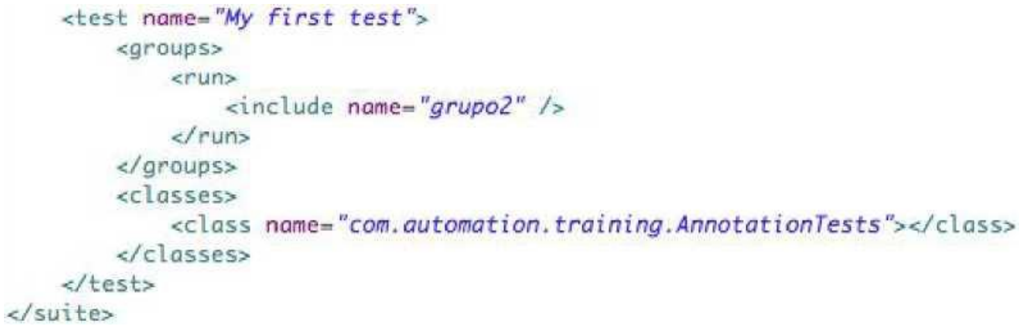


```
@Parameters({ "string1", "string2" })
@Test(groups = {"grupo1" , "grupo2"})
public void testConcact(String a, String b) {
    String resultado = a + b;
    Assert.assertEquals(resultado, "I love test automation");
}
```

As we can see, using the @Parameters annotation we can define them once and then reuse them as many times as necessary in our tests.

3.5.3 - Selecting tests by groups

We can also ask testNG to run only the cases of group 2:



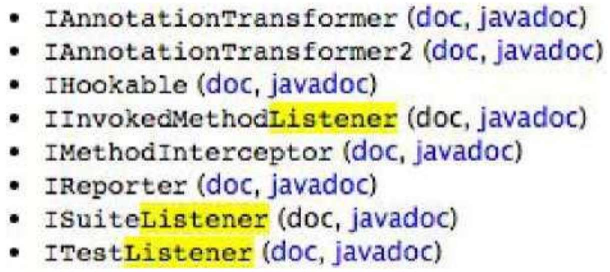
```
    <test name="My first test">
        <groups>
            <run>
                <include name="grupo2" />
            </run>
        </groups>
        <classes>
            <class name="com.automation.training.AnnotationTests"></class>
        </classes>
    </test>
</suite>
```

With this configuration we can then use either include or exclude to configure a suite that includes or excludes the cases that we want.

3.6 - Listeners: What are they and how to use them

A listener is an interface that allows us to modify the behavior of testNG in some way. Basically what we can do is rewrite the behavior of testNG in a certain situation.

There are many of them, to mention a few:



- IAnnotationTransformer (doc, javadoc)
- IAnnotationTransformer2 (doc, javadoc)
- IHookable (doc, javadoc)
- IInvokedMethodListener (doc, javadoc)
- IMethodInterceptor (doc, javadoc)
- IReporter (doc, javadoc)
- ISuiteListener (doc, javadoc)
- ITestListener (doc, javadoc)

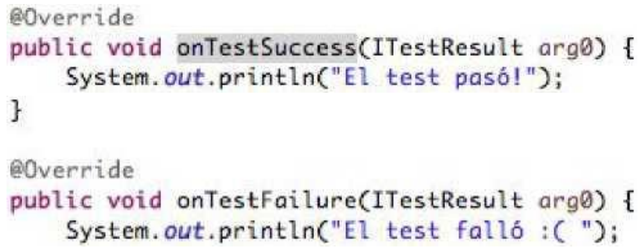
In this example, we are going to create a listener that simply prints something on the screen when a test fails or passes. In order to do so, we are going to create a class in Eclipse, called MyCool Listener that, in turn, implements ITestListener.

```
public class MyCool Li stener implements ITestLi stener’{
```

Then, we must implement the methods of the interface, which are:

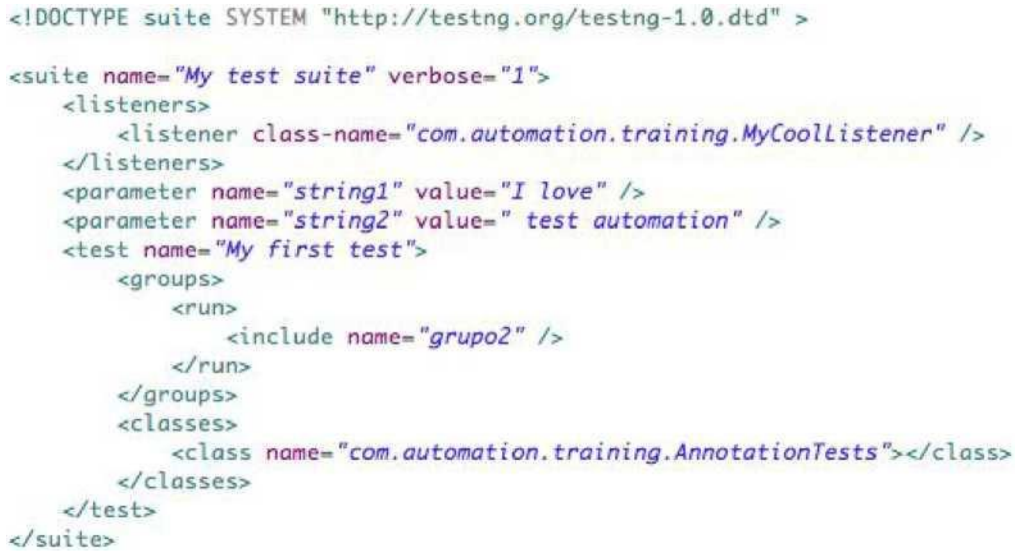
- onFinish
- onStart
- onTestFailedButWithinSuccessPercentage
- onTestFailure
- onTestSkipped
- onTestStart
- onTestSuccess

For our interest, we are going to modify only the methods onTestFailure and onTestSuccess, in the following way:



```
@Override
public void onTestSuccess(ITestResult arg0) {
    System.out.println("El test pasó!");
}

@Override
public void onTestFailure(ITestResult arg0) {
    System.out.println("El test falló :( ");
```

Now, we have to tell testNG to use this listener in our suite; in order to do so, we simply tell it in the suite.xml to do it, in the following way:



```xml
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="My test suite" verbose="1">
    <listeners>
        <listener class-name="com.automation.training.MyCoolListener" />
    </listeners>
    <parameter name="string1" value="I love" />
    <parameter name="string2" value=" test automation" />
    <test name="My first test">
        <groups>
            <run>
                <include name="grupo2" />
            </run>
        </groups>
        <classes>
            <class name="com.automation.training.AnnotationTests"></class>
        </classes>
    </test>
</suite>
```

Now, when running the suite we can see the next console output

- Test passed!
- Test passed!

Now, this example is simple and basically indicates how a listener works, but the uses we can give it are quite extensive, for example:

- Sending an email based on a result
- Updating an external system with the results
- Re-running certain test cases
- Etc.

Summary of the level

In this level we presented two test runner frameworks: TestNG and JUnit and we enlarged on the use of TestNG. We learned how to create a test, the different annotations that TestNG provides and we configured our suite of tests through XML, we filtered our cases by different criteria, we included listeners and finally we ran the tests and recovered their results.