Rex Jones II

Test Next Generation (TestNG) A Powerful Test Framework

Rex Jones II

Intro

Hey Joe, thanks for having me as one of your speakers for 2020 Automation Guild. Also, I want to thank everyone watching this video. I'm going to speak about (TestNG) A Powerful Test Framework.

TestNG stands for Test Next Generation. It's a test framework that makes it easy to write Test Scripts, run Test Scripts, and view Test Results after executing the Test Scripts. We can perform all types of testing. The purpose is to track progress of an Application Under Test (AUT). My goal is to share some Test Patterns that comes built into TestNG. I will show you and not just tell you how to achieve those Test Patterns.

The plan is to demo How To Install TestNG. Next is Annotations then Assertions. After Assertions is Data Driven Testing followed by Dependency Testing and Parallel Testing.

Chp 1 / How To Install TestNG

How To Install TestNG? We can install TestNG using an IDE, Build Tools, Command Line, or download the JAR's. However, I believe Eclipse Marketplace is the fastest way to install TestNG. Go to Help, select Eclipse Marketplace, enter TestNG, and click the Go button. Next, we click the Install button, make sure both checkboxes are checked then click the Confirm button. Finally, we accept the license, click Finish, and Click Install anyway on this Security Warning. The last step is to restart Eclipse. Next is Annotations.

Chp 2 / Annotations

An annotation is information connected to a method. With that connection, an annotation can do a lot of things such as pass data, describe how to pass parameters, and control the execution flow.

There are many annotations and all of them begin with an @ symbol. For now, let's focus on the Configuration Annotations and Test Annotation. The Configuration Annotations are like Pre-Conditions and Post-Conditions. Therefore, the Pre-Conditions start with Before and Post-Conditions start with After. They determine what actions are executed before and after we test.

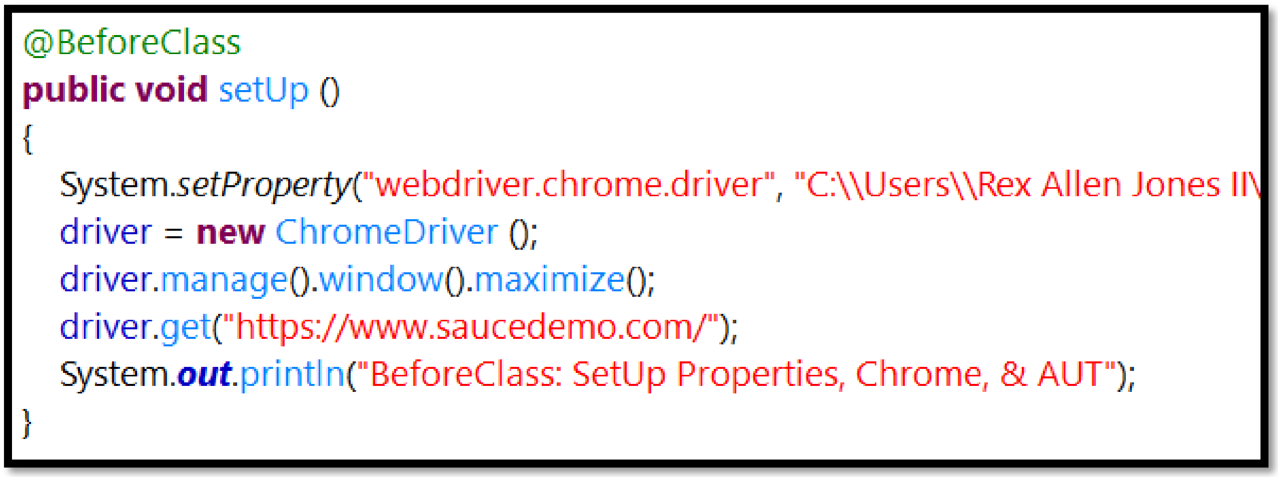
The most used annotations are BeforeClass, AfterClass, BeforeMethod, AfterMethod, and Test. Test represents a Test Method. BeforeClass/AfterClass runs before a test class start and after all Test Methods. BeforeMethod/AfterMethod runs before each Test Method and after each Test Method. Our Test Requirement dictates which pair of annotations we implement. Let's look at 2 Test Requirements.

Our application is Swag Labs by Sauce Labs. It has a Log In page, Products page, Your Cart page, and 2 Checkout Pages. The 1st requirement is to Log In and Log Out. Go back to Eclipse. Our editor shows logIn and logout. We also see setup and teardown. setup is the Pre-Condition and teardown is the Post-Condition. The conditions are logIn and logout.

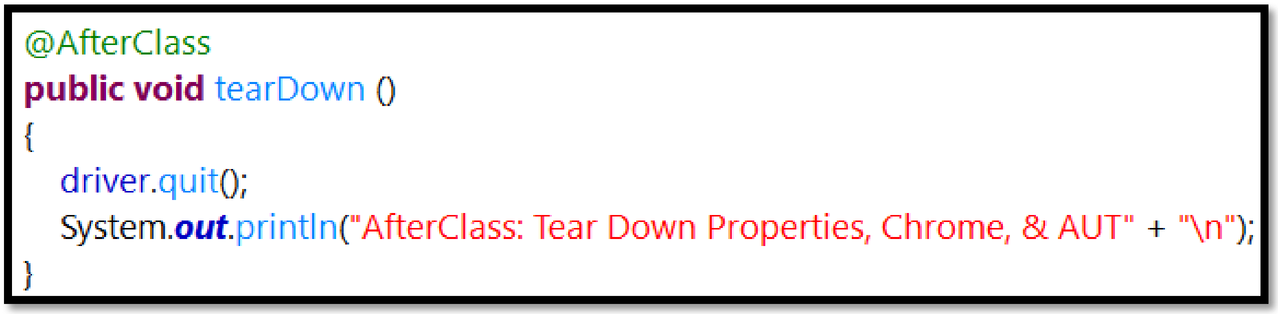
Let's start with BeforeClass, AfterClass, and Test Annotations. We write @ BeforeClass for setup, @ AfterClass for teardown, login @Test, and logout @Test. Every Test will contain a Test Annotation because it performs the main steps of the requirement. Import the Annotations. Notice, setUp and tearDown are at the bottom. We can place annotations anywhere on the editor and they will execute in their defined order.

```
@Test
public void logIn ()
{
   driver.findElement(By.id("user-name")).sendKeys("standard_user");
   driver.findElement(By.id("password")).sendKeys("secret_sauce");
   driver.findElement(By.className("btn_action")).click();
   System.out.println("   Test 1 = Log In");
}
```

```
@Test
public void logOut ()
{
   driver.findElement(By.cssSelector("div.bm-burger-button > button")).click();

   WebDriverWait wait = new WebDriverWait(driver, 5);
   wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("logout_sidebar_link")));

   driver.findElement(By.id("logout_sidebar_link")).click();
   System.out.println("   Test 2 = Log Out");
}
```



```
@BeforeClass
public void setUp ()
{
   System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II
   driver = new ChromeDriver ();
   driver.manage().window().maximize();
   driver.get("https://www.saucedemo.com/");
   System.out.println("BeforeClass: SetUp Properties, Chrome, & AUT");
}
```



```
@AfterClass
public void tearDown ()
{
   driver.quit();
   System.out.println("AfterClass: Tear Down Properties, Chrome, & AUT" + "\n");
}
```

So, here's our Test Script. Before we test, the Test Script will set up the properties, open chrome and access the application. Next, it will log into the application then log out of the application. After logging out, the Test Script will tear down the properties, close Chrome, and close the application. Let's Run. We see logIn and logout PASSED. BeforeClass and AfterClass are the perfect Configuration Annotations for this Test Requirement.

Let's look at the next requirement. We are going to verify 2 separate pages in the same Test Script. Verify the Products page and Checkout Your Information page. This Test Script will use the same pair of Configuration Annotations. BeforeClass sets up the Properties, Chrome, and AUT. AfterClass tears down the properties, Chrome, and AUT. The 2 Test Methods are verifyCheckoutPage and verifyProductsPage. Let's Run. This time we see a failure. verifyCheckoutPage PASSED but verifyProductsPage FAILED. We need the properties, chrome, and AUT to open before verifying the Products Page. This type of requirement calls for the BeforeMethod and AfterMethod Annotations.

Let's add @BeforeMethod for setUp and @AfterMethod for tearDown. Everything else remains the same. The Test Annotation for verifyCheckoutPage and verifyProductsPage.

```
@BeforeMethod
public void setUp ()
{
   System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II\\
   driver = new ChromeDriver ();
   driver.manage().window().maximize();
   driver.get("https://www.saucedemo.com/cart.html");
   System.out.println("BeforeMethod: SetUp Properties, Chrome, & AUT");
}
```

```
@Test
public void verifyCheckoutPage ()
{
   driver.findElement(By.className("checkout_button")).click();
   String labelCheckout = driver.findElement(By.className("subheader")).getText();
   System.out.println("   1. Verify " + labelCheckout);
}
```

```
@Test
public void verifyProductsPage ()
{
   driver.findElement(By.className("btn_secondary")).click();
   String labelProducts = driver.findElement(By.className("product_label")).getText();
   System.out.println("   2. Verify " + labelProducts);
}
```

```
@AfterMethod
public void tearDown ()
{
   driver.quit();
   System.out.println("AfterMethod: Tear Down Properties, Chrome, & AUT" + "\n");
}
```

Let's Run. Now both methods PASSED. The BeforeMethod executed before verifying each page and the AfterMethod executed after verifying each page. Next is Assertions.

Chp 3 / Assertions

The purpose of Assertions is to verify if our Test Passed or if our Test Failed. TestNG provides 2 types of Assertions: A Hard Assert and Soft Assert. A Hard Assert stops executing a Test Method after a failure while a Soft Assert continues executing a Test Method after a failure.

The TestNG package has an Assert class with many Hard Assert methods. Most of the methods are overloaded versions of assertEquals, assertFalse, assertNotEquals, assertNotNull, assertNotSame, assertSame, and assertTrue. The Soft Assert class is located in a different package.

The Assertion Test Script will verify the header (Checkout: Your Information) and place holders (First Name, Last Name). Let's go back to Eclipse with our Test Script. Some of the Test Script has been pre-written to login to the application, select a backpack, click the cart button on the Products page, and click the Checkout button on Your Cart page. Now, we are going to verify the Checkout page.

Hard Assert

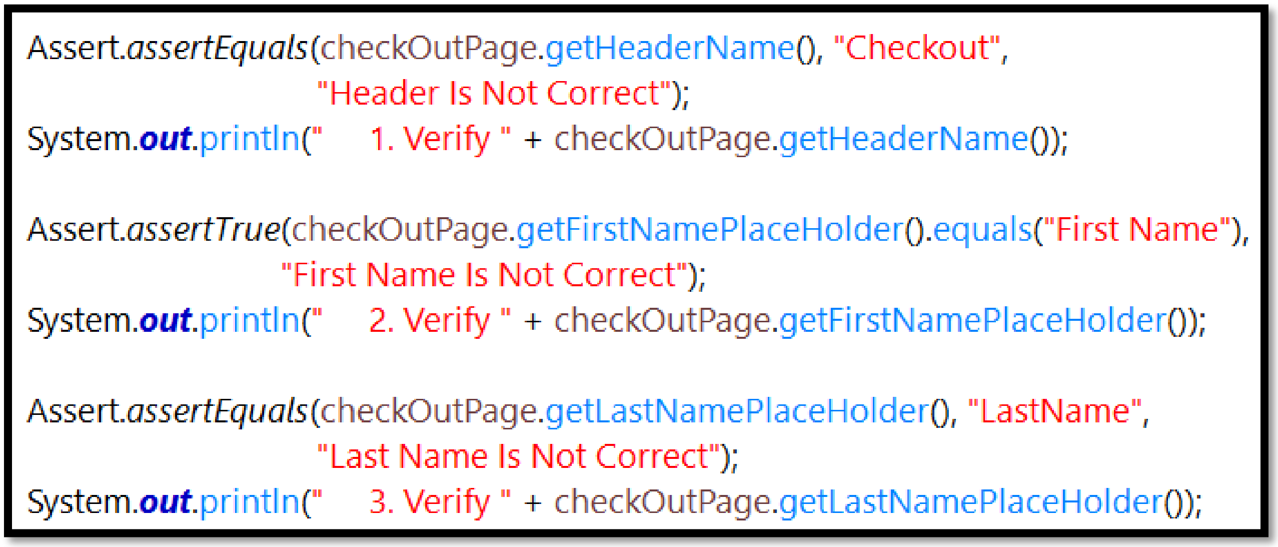
Let's start with Hard Assert by writing the class name Assert.assertEquals(). We see actual and expected. The purpose is to verify if the value coming from the application is the same as what we expect. What's coming from the AUT Checkout page header? checkOutPage.getHeaderName. What do we expect? "Checkout". Recall from the AUT, the header name is Checkout: Your Information. Therefore, this verification will Fail because we only expect Checkout. It's not required but TestNG allow us to add a message that will only be shown if the verification fail. Let's add "Header Is Not Correct". Also, a print statement to verify the Header Name.

My favorite assertion methods are assertEquals and assertTrue. Let's use assertTrue for First Name. Assert.assertTrue(). Now, we see a condition and a message. The condition is on the checkOutPage.get the First Name Place holder and verify if it equals ("First Name"), The message First Name Is Not Correct". Add a print statement to verify the First Name Place Holder.

Let's use 1 more Hard Assert for Last Name.

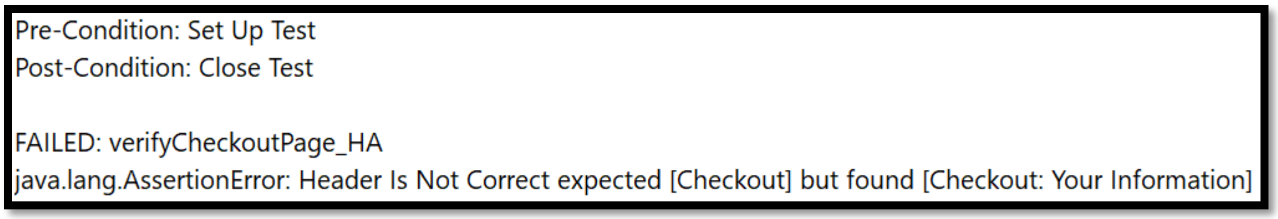
```
Assert.assertEquals(checkOutPage.getLastNamePlaceHolder(), "LastName",
                    "Last Name Is Not Correct");
```

This verification will also Fail because there is not a space between Last and Name. Include a Print statement to verify Last Name.



```
Assert.assertEquals(checkOutPage.getHeaderName(), "Checkout",
                    "Header Is Not Correct");
System.out.println("     1. Verify " + checkOutPage.getHeaderName());

Assert.assertTrue(checkOutPage.getFirstNamePlaceHolder().equals("First Name"),
                  "First Name Is Not Correct");
System.out.println("     2. Verify " + checkOutPage.getFirstNamePlaceHolder());

Assert.assertEquals(checkOutPage.getLastNamePlaceHolder(), "LastName",
                    "Last Name Is Not Correct");
System.out.println("     3. Verify " + checkOutPage.getLastNamePlaceHolder());
```

Let’s Run. The Console shows Pre-Condition: Set Up Test and Post-Condition: Close Test. Where are the other print statements? The other print statements are not available because the first assertion failed then stopped executing our Test. We see an AssertionError: Header Is Not Correct [Checkout] but found [Checkout: Your Information].



```
Pre-Condition: Set Up Test
Post-Condition: Close Test

FAILED: verifyCheckoutPage_HA
java.lang.AssertionError: Header Is Not Correct expected [Checkout] but found [Checkout: Your Information]
```

That’s the problem with Hard Assertions. We have no idea if the other verifications Passed or Failed. Most Test Frameworks only provide a Hard Assert but may have a plug-in for Soft Assert. Let’s look at Soft Assert for the same Test Script.

Soft Assert

First step is to declare SoftAssert with an object softassert equals new SoftAssert. Next, we replace all of the Hard Asserts class names with our object softassert. softassert.assertEquals for Header name. softassert.assertTrue for First Name, softassert.assertEquals for Last Name. There’s 1 more concept for softassert. We must always use assertAll and add it to the end of the Test Method. softassert.assertAll. The purpose of assertAll is to collect all of the Assertion Errors if a verification Fail. Let’s Run. Now we see all of the print statements. As expected, we see 2 asserts failed
Header Is Not Correct expected [Checkout] but found [Checkout: Your Information],
Last Name Is Not Correct expected [LastName] but found [Last Name]

```
SoftAssert softassert = new SoftAssert ();

ProductsPage productsPage = loginPage.login("standard_user", "secret_sauce");
productsPage.selectBackpack();

YourCartPage yourCartPage = productsPage.clickCartButton();
CheckoutPage checkOutPage = yourCartPage.clickCheckoutButton();

softassert.assertEquals(checkOutPage.getHeaderName(), "Checkout",
                        "Header Is Not Correct");
System.out.println("    1. Verify " + checkOutPage.getHeaderName());

softassert.assertTrue(checkOutPage.getFirstNamePlaceHolder().equals("First Name"),
                      "First Name Is Not Correct");
System.out.println("    2. Verify " + checkOutPage.getFirstNamePlaceHolder());

softassert.assertEquals(checkOutPage.getLastNamePlaceHolder(), "LastName",
                        "Last Name Is Not Correct");
System.out.println("    3. Verify " + checkOutPage.getLastNamePlaceHolder());

softassert.assertAll();
```

```
java.lang.AssertionError: The following asserts failed:
      Header Is Not Correct expected [Checkout] but found [Checkout: Your Information],
      Last Name Is Not Correct expected [LastName] but found [Last Name]
```

SoftAssert is powerful because it allows us to see the results of all verifications whether they Pass or Fail. Next is Data Driven Testing.

Chp 4 / Data Driven Testing

Data Driven Testing is when we store information in a data source then use that information as input for a Test Method. The data can be any type and we can use different types of data sources. I'm going to use the DataProvider Annotation as the data source.

We are going to log into Swag Labs using these 4 usernames. All of the usernames have the same password secret_sauce. However, locked_out_user is the only username that will not log into Swag Labs.

The benefit of Data Driven Testing is to separate logic from data. We can execute 1 Test Script which has the logic and use multiple sets of data. Our Test Script is verifyLogin and logInData contains our data sets. The key is to connect both of these methods together. We connect them using the DataProvider Annotation. All DataProvider Annotations have an optional (name = “”) attribute. If we choose not to use the name attribute, then the method name logInData becomes the default DataProvider name. However, it’s best to use the name attribute. Let’s call this DataProvider login-provider.

```
@DataProvider (name = "login-provider")
public static String [] [] logInData ()
{
   String [] [] data = new String [4] [3];

   data [0] [0] = "standard_user";          data [0] [1] = "secret_sauce";   data [0] [2] = "Pass";
   data [1] [0] = "locked_out_user";        data [1] [1] = "secret_sauce";   data [1] [2] = "Fail";
   data [2] [0] = "problem_user";           data [2] [1] = "secret_sauce";   data [2] [2] = "Pass";
   data [3] [0] = "performance_glitch_user"; data [3] [1] = "secret_sauce";  data [3] [2] = "Pass";

   return data;
```

Let’s go to our Test Method which has the logic. If both methods were in the same class, we could write a (dataProvider) attribute = the DataProvider Annotation’s name “login-provider” and be finished. But both methods are in separate classes so we must include the dataProviderClass attribute. The dataProviderClass = the class containing the DataProvider Annotation LogInDataProvider.class.

```
@Test  (dataProvider = "login-provider", dataProviderClass = LogInDataProvider.class)
public void verifyLogIn (String usename, String password, String status)
{
   ProductsPage productsPage = loginPage.login(usename, password);

   System.out.println("   Username = " + usename + "\n" +
                      "   Password = " + password + "\n" +
                      "   Status = " + status);

   Assert.assertTrue(productsPage.getProductLabel().equals("Products"),
                     "Please Enter Valid Login Credentials");
```

Let’s Run. standard_user PASSED / locked_out_user FAILED / problem_user PASSED / performance_glitch_user PASSED. We see 1 Failure. Each data contains their own iteration. 3 PASSED / 1 FAILED. Next is Dependency Testing.

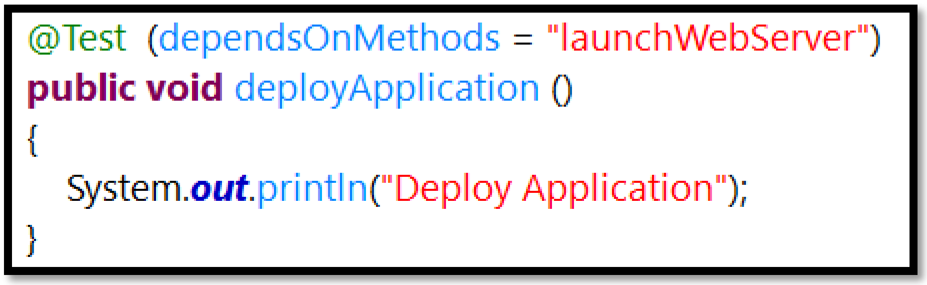
Chp 5 / Dependency Testing

Dependency Testing is when 1 Test Method depends on 1 or more Test Methods. This feature is important because there are times when we should not execute a Test if the previous Test Failed. TestNG has 2 attributes for Dependency Testing. Those attributes are dependsOnMethods attribute and dependsOnGroups attribute. Let’s look at the dependsOnMethods Attribute.

First, I’m going to run this program without the dependsOnMethods Attribute so you can see the misleading Test Results. Do you this priority Attribute = 3? The purpose of priority is to execute each Test Method in a defined order. By default, TestNG executes the Test Methods in alphanumeric order. setUpApplication will fail because we are accessing the WrongURL and assertTrue is false. The Plan is to execute all 5 tests after setting up our application. Let’s Run.

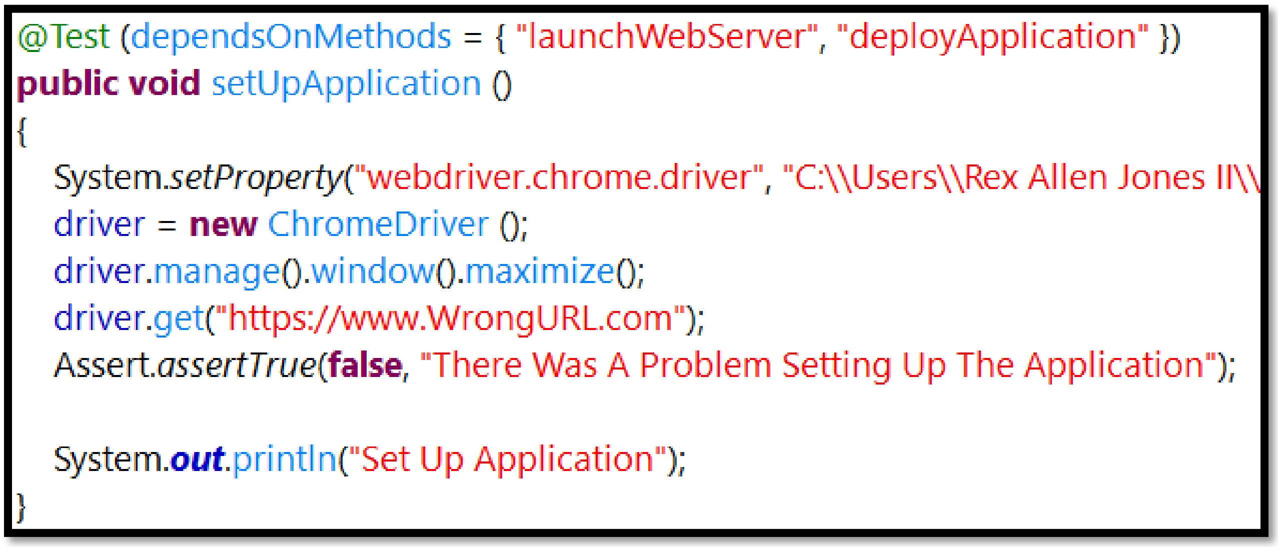
As expected, WrongURL.com but what’s not expected is a cascade of failures. LaunchWebServer PASSED, deployApplication PASSED but setUpApplication FAILED. We see the Test Results show all 5 tests FAILED. How did each test FAIL if we never set up the application? That’s misleading and that’s why it’s beneficial to use Dependency Testing.

Here’s the same Test Script with dependsOnMethods. Since it’s an attribute, we write (dependsOnMethods =) then the method name it depends on. This method deployApplication depends on “launchWebServer”.



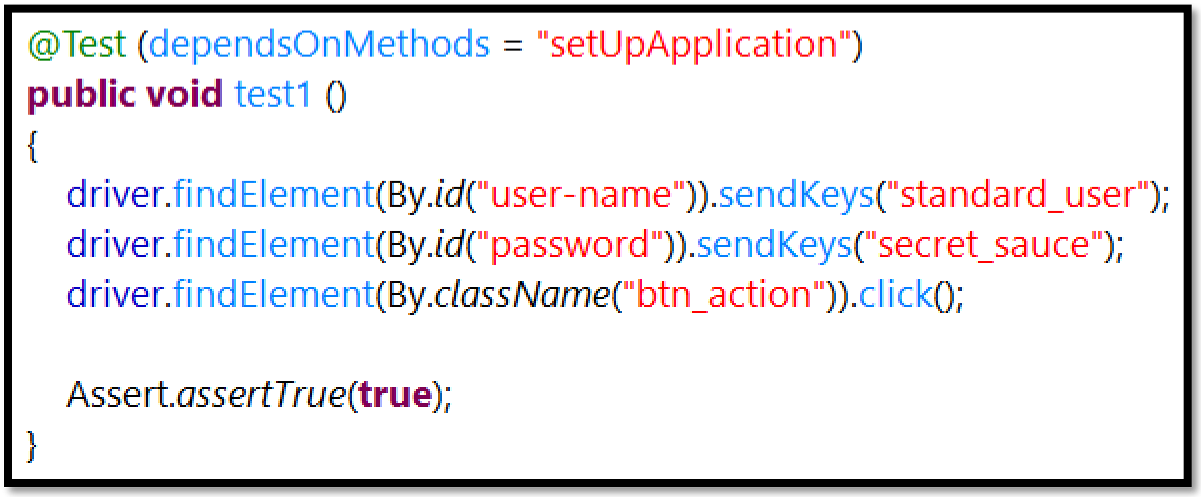
```
@Test  (dependsOnMethods = "launchWebServer")
public void deployApplication ()
{
   System.out.println("Deploy Application");
}
```

We can also write more than 1 method name. setUpApplication only requires 1 method it depends on but let’s write 2 (dependsOnMethods = {“launchWebServer”, deployApplication}).



```
@Test (dependsOnMethods = { "launchWebServer", "deployApplication" })
public void setUpApplication ()
{
   System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II\\
   driver = new ChromeDriver ();
   driver.manage().window().maximize();
   driver.get("https://www.WrongURL.com");
   Assert.assertTrue(false, "There Was A Problem Setting Up The Application");

   System.out.println("Set Up Application");
}
```

All 5 tests depend on setUpApplication. Write (dependsOnMethods = “setUpApplication“). The other 4 tests already have dependsOnMethods equal to setUpApplication.



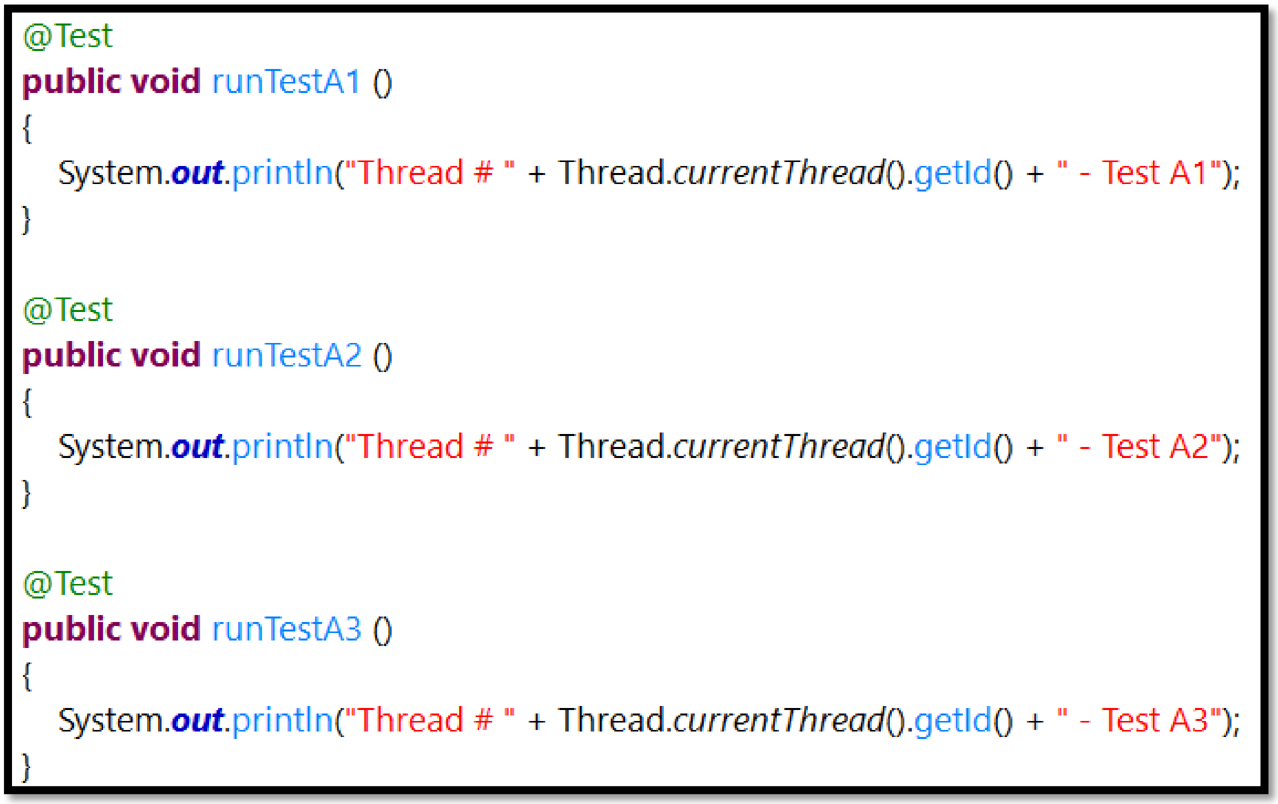
```
@Test (dependsOnMethods = "setUpApplication")
public void test1 ()
{
   driver.findElement(By.id("user-name")).sendKeys("standard_user");
   driver.findElement(By.id("password")).sendKeys("secret_sauce");
   driver.findElement(By.className("btn_action")).click();

   Assert.assertTrue(true);
}
```

Let’s Run. WrongURL.com. Now, we see 1 FAILED, and 5 SKIPPED. That’s correct because test 1 – 5 were never executed so they should show SKIP. The Test Results show 2 PASSED, 1 FAILED, and 5 SKIPS. Next is Parallel Testing.

Chp 6 / Parallel Testing

The advantage of Parallel Testing is maximum speed. We can run each test in a separate thread. Thread is a path when executing our program. At runtime, in the xml file, TestNG must be configured to run in parallel mode. In our Test Script, we set the parallel attribute to true.

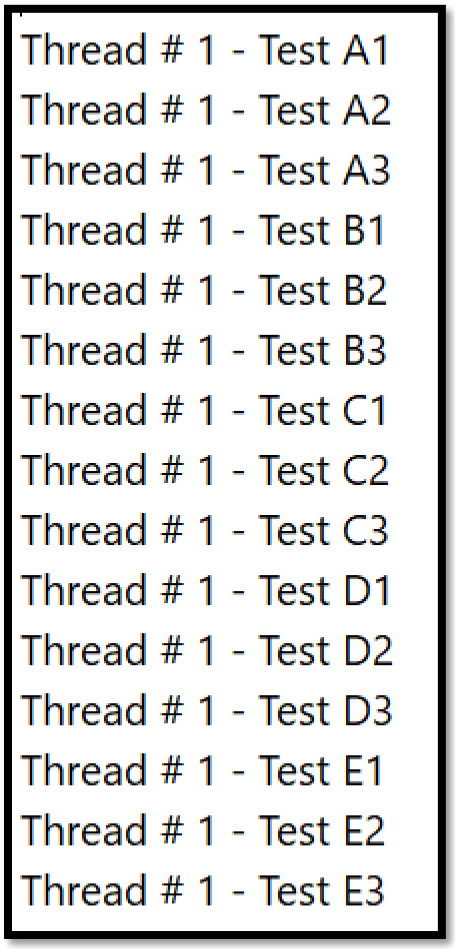
Here's 5 Test Scripts that contains 3 Test Methods. Let's look at ParallelTestRun_A. We see a Test Method for runTestA1, runTestA2, and runTestA3. All 5 Test Scripts have the same kind of Test Methods from A to E.



```
@Test
public void runTestA1 ()
{
   System.out.println("Thread # " + Thread.currentThread().getId() + " - Test A1");
}

@Test
public void runTestA2 ()
{
   System.out.println("Thread # " + Thread.currentThread().getId() + " - Test A2");
}

@Test
public void runTestA3 ()
{
   System.out.println("Thread # " + Thread.currentThread().getId() + " - Test A3");
}
```

Single Thread Mode

First, I will run each test in a single thread mode with no parallel. You can create an xml file by right clicking the class, select TestNG, then convert to TestNG. Each class has been divided into 3 test tags. Test 1, Test 2, and Test 3. Therefore, there are 15 tests total. Let's Run. All 15 Tests ran in the same thread - Thread 1.

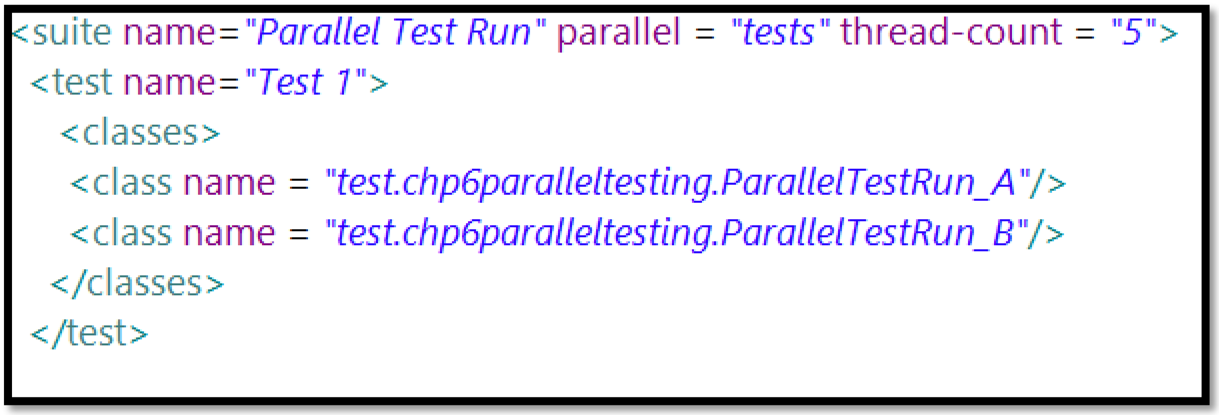


```
Thread # 1 - Test A1
Thread # 1 - Test A2
Thread # 1 - Test A3
Thread # 1 - Test B1
Thread # 1 - Test B2
Thread # 1 - Test B3
Thread # 1 - Test C1
Thread # 1 - Test C2
Thread # 1 - Test C3
Thread # 1 - Test D1
Thread # 1 - Test D2
Thread # 1 - Test D3
Thread # 1 - Test E1
Thread # 1 - Test E2
Thread # 1 - Test E3
```

There are 2 parallel modes for TestNG. The modes are tests and methods.

Parallel Test Mode

Let's start with the tests mode. In this mode, all of the methods in a <test> tag will run in their own thread. Therefore, we will see 3 threads because there are 3 <test> tags. In the suite tag, add parallel = "tests" then thread-count = "5".

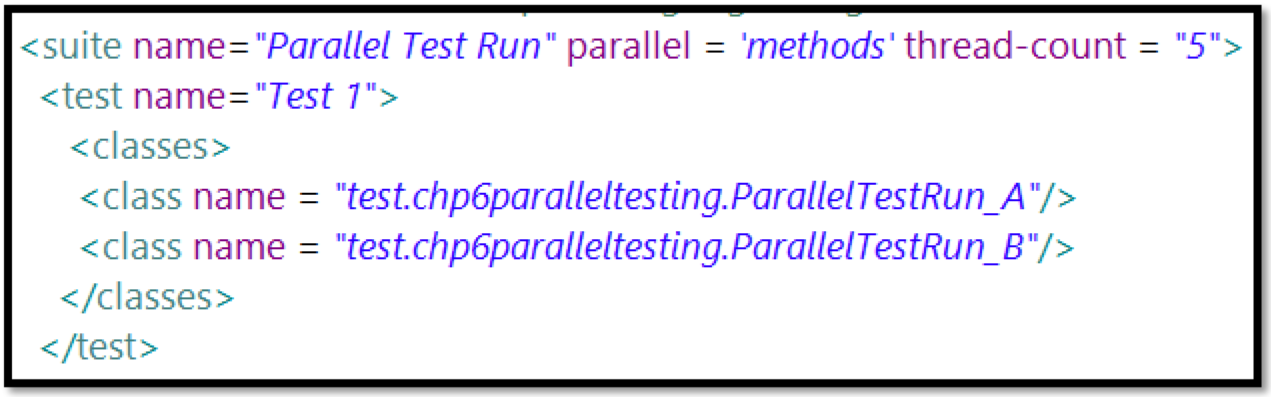


```xml
<suite name="Parallel Test Run" parallel = "tests" thread-count = "5">
 <test name="Test 1">
   <classes>
    <class name = "test.chp6paralleltesting.ParallelTestRun_A"/>
    <class name = "test.chp6paralleltesting.ParallelTestRun_B"/>
  </classes>
 </test>
```

For this scenario, thread count doesn't matter because it's only 3 tests. Let's Run. We see 3 threads for 15 tests. Parallel Testing helps speed up our execution because there's more than 1 thread.

Parallel Methods Mode

Now, let's look at the methods mode. In this mode, each Test Method will run in its own thread. Add parallel = "methods" with the same thread-count = "5".

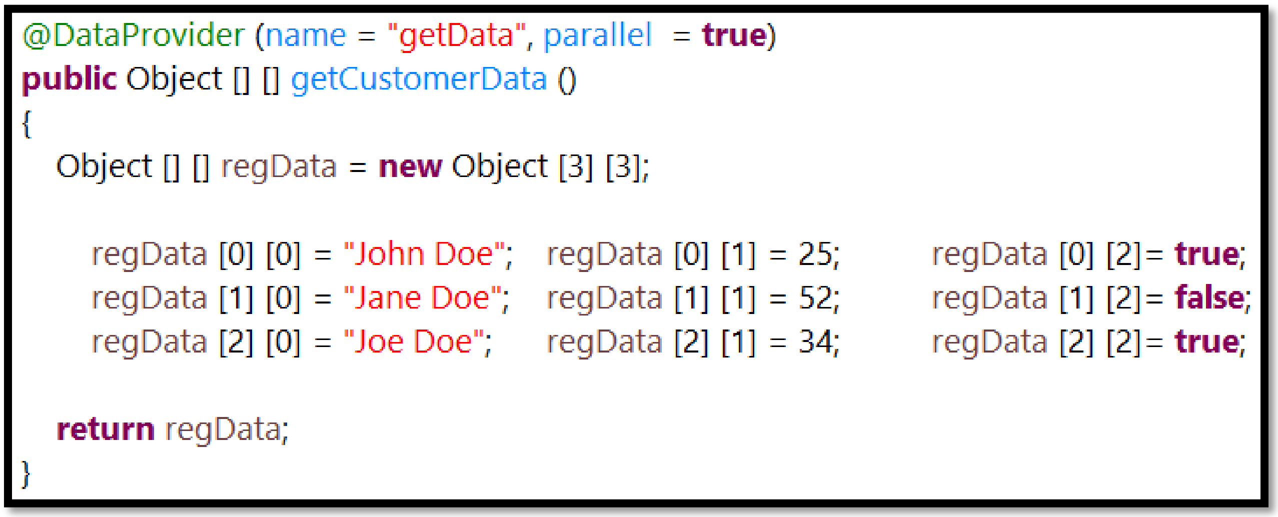


```xml
<suite name="Parallel Test Run" parallel = 'methods' thread-count = "5">
 <test name="Test 1">
   <classes>
    <class name = "test.chp6paralleltesting.ParallelTestRun_A"/>
    <class name = "test.chp6paralleltesting.ParallelTestRun_B"/>
  </classes>
 </test>
```

Notice in the xml file, I right click then select Run As TestNG Suite. There's a thread for almost every Test Method. Sometimes, the same thread can show up more than 1 time. It will show up again if a thread opens back up after executing a test.

Parallel – Drive Data Testing

Here's one more example of Parallel Testing using the DataProvider Annotation. We see 3 data sets consisting of name, age, and true – false values. If we want to run each data set in its own thread then add parallel = true to the DataProvider Annotation.



```
@DataProvider (name = "getData", parallel  = true)
public Object [] [] getCustomerData ()
{
   Object [] [] regData = new Object [3] [3];

      regData [0] [0] = "John Doe";   regData [0] [1] = 25;      regData [0] [2]= true;
      regData [1] [0] = "Jane Doe";   regData [1] [1] = 52;      regData [1] [2]= false;
      regData [2] [0] = "Joe Doe";    regData [2] [1] = 34;      regData [2] [2]= true;

   return regData;
}
```

Let's Run. There are 3 threads for each data set: 14, 15, and 16.

Wrap-Up

That wraps up TestNG. We covered How To Install TestNG, Annotations, and Assertions. Annotations is data connected to a method. Assertions verify if our Test Passed or Failed. There are 2 types of Assertions. Hard Assert and Soft Assert. A Hard Assert stops execution in a Test Method after a failure while a Soft Assert continues execution after a failure.

Then we looked at 3 Test Types: Data Driven Testing, Dependency Testing, and Parallel Testing. Data Driven Testing separates logic from data. It can execute 1 Test Script with multiple sets of data. Dependency Testing produces an accurate report when a Test Method depends on 1 or more Test Methods. If the previous Test Method Fail then the next Test Method is Skipped. Last is Parallel Testing which speeds up execution by running each Test in a separate thread. Thanks for watching (TestNG) Test Next Generation – A Powerful Test Framework. The automation code and transcript will be available on https://github.com/RexJonesII