Author: Alexander Boffin

Version:18-01-17

**Selenium WebDriver documentation**

**Inhoudsopgave**

[Before Starting 3](#_Toc472059572)

[Basics Java 3](#_Toc472059573)

[Coding Standards: 3](#_Toc472059574)

[Shortcut to Method Creation: 4](#_Toc472059575)

[IntelliJ: 4](#_Toc472059576)

[Extra Sites To Run Tests On: 4](#_Toc472059577)

[Lab1: Overview 5](#_Toc472059578)

[0)Add Elements To The pom-file. 5](#_Toc472059579)

[A)Creating A New Instance Of The Firefox Driver 5](#_Toc472059580)

[B)Opening The URL To A Website: 6](#_Toc472059581)

[C)Selecting An Element On The Page 6](#_Toc472059582)

[D)Clicking On An Element 7](#_Toc472059583)

[E)Wait 7](#_Toc472059584)

[F)Type/Send Text 8](#_Toc472059585)

[G)Select And Store Text From The Page (Store Command) 9](#_Toc472059586)

[H)Verify If Text Is Correct (Assertions) 9](#_Toc472059587)

[I)Cleaning Up/Closing The Driver 9](#_Toc472059588)

[J)Rollup=>Method Creation 9](#_Toc472059589)

[How to execute the driver over different methods 10](#_Toc472059590)

[Using JUnit 11](#_Toc472059591)

[Using TestNG 12](#_Toc472059592)

[Example GoogleSearch: 13](#_Toc472059593)

[Lab02: Debugging 15](#_Toc472059594)

[Lab03: Identifying Elements 16](#_Toc472059595)

[The XPath Guide: 16](#_Toc472059596)

[Selenium IDE Remark: 18](#_Toc472059597)

[Lab04: Selenium Syntax 19](#_Toc472059598)

[A)Assert The Title 19](#_Toc472059599)

[B)Operators 19](#_Toc472059600)

[C)EXTRA: Handy, But Not Essential Tricks 20](#_Toc472059601)

[Lab05: Conditions And Loops 22](#_Toc472059602)

[A)While Loops 22](#_Toc472059603)

[B)The For Loop 22](#_Toc472059604)

[C)if, else if And else 23](#_Toc472059605)

[Lab 06: Data Driven Testing 24](#_Toc472059606)

[A)CSV Files 24](#_Toc472059607)

[B)XML Files 26](#_Toc472059608)

[C)JSON FILE 28](#_Toc472059609)

[Lab 08: Extras For WebDriver 30](#_Toc472059610)

[A)Using Different Web Browsers. 30](#_Toc472059611)

[B)Selenium Grid (Doing Parallel Testing 30](#_Toc472059612)

[Setting Up The Hubs And Nodes 30](#_Toc472059613)

[Method 1: To Make Tests Run In Parallel Using JUnit 33](#_Toc472059614)

[Method2: Run Tests In Parallel Using TestNG 36](#_Toc472059615)

[Lab 09: Advanced Stuff 40](#_Toc472059616)

[iFrames 40](#_Toc472059617)

[See If There Are iFrames On The Page: 40](#_Toc472059618)

[Switching To The iFrame Using WebDriver 41](#_Toc472059619)

[Switch Back To A Previous Frame 41](#_Toc472059620)

[The Concept Of Nested Frames: 42](#_Toc472059621)

[Parameterization With TestNG=>Data Driven Testing 43](#_Toc472059622)

[(1)Using The Parameter Annotation And The xml-file 43](#_Toc472059623)

[(2a)Parameters Using The DataProvider (Default) 44](#_Toc472059624)

[(2B) Parameters Using The DataProvider (Method) 44](#_Toc472059625)

[(2C) Parameters Using The DataProvider (ITest) 45](#_Toc472059626)

[Handeling Cookies In Selenium Webdriver 49](#_Toc472059627)

[(1)Storing The Cookie Info: 49](#_Toc472059628)

[(2)Using The Stored Cookie To Log Into The Application: 50](#_Toc472059629)

[Alert & Popup Handling 52](#_Toc472059630)

[(A)Alerts 52](#_Toc472059631)

[(B)Popup Windows 53](#_Toc472059632)

[OOP (Object Oriented Programming) 55](#_Toc472059633)

[Inheritance 55](#_Toc472059634)

[Polymorphism 55](#_Toc472059635)

[Abstraction 55](#_Toc472059636)

[Encapsulation 55](#_Toc472059637)

[Page Object Pattern (POM) By Using PageFactory 56](#_Toc472059638)

[(1)Regular POM 56](#_Toc472059639)

[(2)Pom Using PageFactory 58](#_Toc472059640)

[Cumcumber 61](#_Toc472059641)

# Before Starting

## Basics Java

This text is inspired by the Selenium IDE Labs. But before moving on to the Labs it is recommended you get familiar with the basics of the java programming language. There are several good sites to start with. take a look at the following webpages and try the one you think best suits you:

* <http://docs.oracle.com/javase/tutorial/>

=>Is from oracle itself. And is one of the most complete documentations. If using this one, I’ll recommend looking primarily at “Getting Started” and “Learning the Java Language”. The rest can be used to improve yourself later on.

* <http://www.learnjavaonline.org>

=>A site that learns the basics in an interactive way. To get a grip with the essential elements by looking at “Learn the basics”.

* <http://beginnersbook.com/java-tutorial-for-beginners-with-examples/>

This is a nice starting point whit a lot of example programs.

* <https://www.tutorialspoint.com/java/>

=>Best used as an extra resource. The site contains a lot of information and can help if you have trouble with particular concepts. Tutorialpoint itself is a nice resource to expand your knowledge of different topics. I would recommend to browse around in your spare time.

* <http://www.guru99.com/java-tutorial.html>

This is a site with simplified explanations. Like Tutorialpoint it is a site best used if you are stuck on something and need clarification. The site has a lot of nice information on different topics surrounding testing. Take a look and see what you can find to polish your skills.

* <http://docs.oracle.com/javase/tutorial/>

=> You can use this site to test your understanding and find new ways of doing stuff. It’s just a fun site to know, not essential for getting a grips with the WebDriver or Java.

You don’t become good in programming by just looking or doing exercises just once. Becoming good at something means doing exercises on the subject regularly and experimenting with what you have learned (it’s comparable to a marathon, not a sprint or dash to the finish line). A few sites are handy to motivate you to keep on coding. The following sites can be used outside of your worktime if you want:

* <https://codecombat.com>
* <https://www.codingame.com>

## Coding Standards:

There is a separate document “java Coding standards” that goes more in detail about this topic. But here are some elements to start with:

It is good practice to use elements as // and /\* \*/ to place commentary and explanations in your programs. This will seriously reduce the time spend later on figuring out what your intentions were when writing it. Using the comments as heathers can break your program in easier to overlook compartments. These help to give structure to your code.

You can use // for comments, while using the /\* \*/ like

*/\* ----------------------------------------------------------  
Heather  
\*-----------------------------------------------------------\*/*

or

*/\* ------------------------Heather-------------------------\*/*

System.***out***.println(**"Personalised What is going on here"**);

It is always handy to know that your program is doing. For this the print-method can be helpful. Try to use them in such a way that the program tells you what is going on. This can be done by displaying where the program is (“I am at part…”) or feed it variables that need to be used. But don’t overdo it to much!

Because other people are going to be reading your file use clear, meaningful names for variables and methods. It can be compelling to use funny names, but keep the urge under control. Remember: “Your program needs to be readable by your colleagues”

## Shortcut to Method Creation:

It can be bothersome to completely type the following syntax:

**public static void** main(String[] args) {}

A shortcut is to type psvm and then press shift(->). This will quickly create the syntax

After this you can modify the method to the one you need (like changing the name and remove “void” or static).

## IntelliJ:

To get more information on the IntelliJ IDE, you can use the next link:

<https://www.jetbrains.com/help/idea/2016.3/meet-intellij-idea.html>

## Extra Sites To Run Tests On:

* Guru99 has set up a “bank website” where you can practice on.

<http://demo.guru99.com/v4/>

You just need to generate access by giving in your Email address. This will send a UserID and Password to you that is active for 20 days.

* The site <http://demo.avactis.com> offers different E-commerce stores where you can practice upon.

# Lab1: Overview

Here are some elements to read/do before you can begin the “Google search” exercise

## 0)Add Elements To The pom-file.

In the “Set up WebDriver” file the need to change the pom file is already stated. You needed to add the following:

<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.seleniumhq.selenium</**groupId**>  
 <**artifactId**>selenium-java</**artifactId**>  
 <**version**>3.0.1</**version**>  
 </**dependency**>

<**dependency**>  
 <**groupId**>org.seleniumhq.selenium</**groupId**>  
 <**artifactId**>selenium-firefox-driver</**artifactId**>  
 <**version**>3.0.1</**version**>  
 </**dependency**>

<**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**version**>4.12</**version**>  
 </**dependency**>  
</**dependencies**>

The first dependency is for letting selenium work with the java commands in the IDEA. The second is necessary to let it work with the Firefox driver. The last is a unit testing framework necessary to do several of the testing steps. (Don’t forget to see if the elements in red are the latest version available).

Another framework that we are going to use is TestNG. The framework is a more extensive version of JUnit (so it covers all categories of tests, not just the unit tests). For a more information over how the frameworks differ see the document “JUnit VS TestNG”. The element that need to be added to the pom file to get this framework is:

<**dependency**>  
 <**groupId**>org.testng</**groupId**>  
 <**artifactId**>testng</**artifactId**>  
 <**version**>RELEASE</**version**>  
 <**scope**>test</**scope**>  
</**dependency**>

REMARK: Always take care not to mix the annotations (see further in the text) of the TestNG and JUnit. This can result in test failures (These can be easily overlooked when debugging). A more thorough comparison of the two frameworks is in the document “JUnit vs TestNG” (can be read later on to get more information about the two frameworks).

## A)Creating A New Instance Of The Firefox Driver

With geckodriver in “lib” (see the “Setting Up WebDriver” document)

System.*setProperty*(**"webdriver.gecko.driver"**, **"lib/geckodriver"**);

WebDriver driver = **new** FirefoxDriver();

Since Selenium 3 all browsers have their own driver. For Firefox we need the geckodriver to invoke it.

## B)Opening The URL To A Website:

driver.get(**"http://www.google.be"**);

## C)Selecting An Element On The Page

TIP: Next to firebug it can be handy to use the select function of the Selenium IDE. This give you an idea what can be used to target the element (see Lab 3). When selecting a text field for clearance it helps to choose a good target (for some drivers this is a must).

There are several ways to select an element. (like with the Selenium IDE).

By ID

driver.findElement(By.*id*(**"ID name"**));

By Classname

driver.findElement(By.*classname*(**"classname"**));

By <Tagname>

driver.findElement(By.*tagName*(**"tagname"**));

By name

driver.findElement(By.*name*(**"naam"**));

By link text <a href= “ ”>Linktext</a>

driver.findElement(By.*linkText*(**"linktext"**));

By partial link text

driver.findElement(By.*partialLinkText*(**"partial text"**));

By CSS Selector =>using a combination of an element selector and an element value.

driver.findElement(By.*cssSelector*(**"(\*)"**));

**(\*)can be;**

**-Tagname#id**

**-Tagname.classname**

**-Tagname[attribute=’atribute name’]**

**-Tagname[attribute1=’atribute name’] [attribute2=’atribute name’]**

**Preforming partial matches:**

**-[id^=’start text’]**

**+>^means “Starting with”**

**-[id$=’tekst at the end’]**

**+>$means “Ending with”**

**-[id\*=’tekst in the middle]**

**+>\*means “contains”**

By XPATH (using the HTML DOM structure to find the location)

driver.findElement(By.*xpath*(**"(\*)"**));

(\*)can be;

-Absolute path: starting at the root node

Initiation with “/”=>“/…/endelement”

-Relative path: search anywhere on the page structure

Initiation with “//”=> “//endpath”

Elements that can help in the selection process:

+Using an index “”//element[int number]

+Using an atribute ”//element[@atribute=’value’]”

The elements give here are good to start with. A more extensive guide to use XPath is placed in Lab 3 (see further on).

Small remark: You can place the element locators at the start of the class (as global variables) if working with multiple methods (see later on with “rollup”). If there is a change in the page-structure it is easier to change the elements there, than looking throughout the entire script to change them. This can be done by using:

By namegiventolocator=By.locatortype=”locatorelement”

Example:

By wikitextongoogle=By.cssSelector("span.st");

This can then be sused as:

driver.findElement(namegiventolocator)

## D)Clicking On An Element

driver.findElement(By.name("btnG")).click();

First you find the element using a method from part C. Then you add the .click() command. You’ll see that most configurations are built like this.

## E)Wait

This is the equivalent of the ClickandWait command in Selenium IDE. There are 3 types of waits:

1. Implicit waits: at start of your Javafile. The driver will keep polling a certain amount of time until it has found the element you want to use in the test. This needs to be set only once, at the start of the class. If the elements aren’t found before the set time an “ElemenyNotVisibleException” will be trown.

driver.manage().timeouts().implicitlyWait(2, TimeUnit.***SECONDS***);

1. Explicit wait: setting a separate time on a required element. This is more flexible than setting an exceptionally high implicit wait. First the driver will wait the explicit wait time and then the implicit wait time, before throwing the “elementNotVisibleException”. I would recommend setting it when you go to a new page. If the page loads very slowly the test fails because the intended element couldn’t be located immediately. So first try the debugger to go through the different steps in the program. It can be that here the file goes through the steps without failing. This gives an indication that a wait has to be placed somewhere.

a)Set a wait at the start of the file=>can be referenced to several times

WebDriverWait wait= new WebDriverWait(driver,10);

b)The element in step a can be called upon whenever needed. There are several ways of doing it. My most used form I use is:

wait.until(ExpectedConditions.*visibilityOfElementLocated*(By

.*cssSelector*(**"span.st"**)));

Other methods that are commonly used are:

visibilityOfAllElementsLocatedBy()

alertIsPresent()

titleIs()

titleContains()

visibilityOf()

elementToBeClickable()

textToBePresentInElement()

textToBePresentInElementLocated()

textToBePresentInElementValue()

visibilityOfAllElements()

elementSelectionStateToBe()

elementToBeSelected()

frameToBeAvaliableAndSwitchToIt()

invisibilityOfTheElementLocated()

invisibilityOfElementWithText()

presenceOfAllElementsLocatedBy()

presenceOfElementLocated()

More information about these and other classes that can be used can be found at:

<https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/support/ui/ExpectedConditions.html>

(!)If a particular page is loading very slowly set a separate/new WebdriverWait there instead of changing the one in the start of the file.

1. Fluent wait: here the frequency of the polling can be set (and the specific type of exception that needs to be ignored). It isn’t as much used as explicit wait and is only mentioned here for completeness of the information given. To initialize use:

Wait wait = new FluentWait(driver)

.withTimeout(30, SECONDS)

.pollingEvery(5, SECONDS)

.ignoring(NoSuchElementException.class);

Then you can set the element that needs to be used in the test and needs to be wait on by:

WebElement neededelement = wait.until(new Function() {

public WebElement apply(WebDriver driver) {

return driver.findElement(By.id("foo"));

}

});

neededelement.click();

## F)Type/Send Text

Remark: only the sendkeys method is available here (just add text to a field). You call the method by adding .sendKeys(“type”) to the selected element:

driver.findElement(By.*id*(**"sb\_ifc0"**)).sendKeys(“type”);

If you want to add new text to the field, without keeping the text present, the text field it needs to be cleaned up (for example if you want to do a new search). This can be done several ways:

The fastest/easiest method to clear the field is by using .clear():

driver.findElement(By.*id*(**"lst-ib"**)).clear();

Another method is

driver.findElement(By.*id*(**"lst-ib"**))

.sendKeys((Keys.Control,”a”), Keys.Delete);

If it doesn’t want to work use debugger to see if the field is properly cleared and new text is added. It can be that the right element isn’t selected. Then you can try to use the select button of the Selenium IDE.

## G)Select And Store Text From The Page (Store Command)

For doing getting text from the page you use .getText():

driver.findElement(By.*cssSelector*(**"span.st"**)).getText();

It is also possible to store specific elements of an element using .getAttribute(“Attribute”)

## H)Verify If Text Is Correct (Assertions)

Remark: Strings are treated differently in java than in most other programming languages. They are treated as objects and so “IsTheSame”==”IsTheSame” will return False (while 0==0 will return true). This is because they are two are different objects. To see if they are logical equals we need something else:

Option 1: to see if they are exactly the same

Assert.*assertTrue*(actual, expected);

Option 2:

Assert.*assertTrue*(actual.equals(expected));

EXTRA: To see if it contains a particular element

Assert.assertTrue(TextInWiki.contains(“partial text”));

## I)Cleaning Up/Closing The Driver

At the end of a program it can be handy to close the driver window in the program itself. Otherwise you’ll need to do it manually after running the test. You can let the program close the window using:

driver.quit();

It is always nice if something else cleans up for you.

## J)Rollup=>Method Creation

In the Senenium IDE it was possible to add a file to the program, that had a function we could execute in different test cases (reusing that code over and over again). Instead of using an external file here, you can create a method inside the Class file (in OOP we will extend this into using methods made in other classes). This is like breaking a big whole into smaller manageable pieces. Each piece can be called multiple times. This means you don’t have to rewrite the same code again if needed.

A method is placed within a class and has the following structure:

public static int methodName(variable1, ….) {

// body

}

These can then be called by other methods by using methodName(variable1, ….).

In the text here you have seen several examples of this like *assertTrue*(actual, expected).

Within a class there are two types of variables:

1. Global variables: these are places at the start of the class and outside an method. They exist for the entire lifespan of the program running in the class. They can be accessed by any method in the class.
2. Local variables: These variables are declared inside the body of a method and have a limited lifespan (only while method is being used). The variables can’t be accessed by other functions.

But if you try to use the same driver over different functions there are several ways to do this while avoiding the hated error messages.

### How to execute the driver over different methods

You can make different methods of the steps to be taken, the driver setup and the driver breakdown. If you don’t make the methods static you can use the WebDrive if you place him as a field member (just below the class-notation and is thus shared by the different methods, that aren’t static). When you have this you can place the But if you do this, you’ll see that you can’t execute the test itself. To remedy this, you create a test that needs to be run by placing on top of it the @Test using the JUnit or TestNG framework. This test can then be run to see if it works. An example of how this structure might look is:

WebDriver driver;

/\*This method can be executed\*/

@Test

public void tryingItThisWay() {

theTest();

}

public void setDriver() {

System.setProperty("webdriver.gecko.driver", "lib/geckodriver");

driver = new FirefoxDriver();

}

public void cleanup(){

driver.quit();

}

public void theTest(){

setDriver();

//Steps for the automation

cleanup();

}

### Using JUnit

This calls the method named assertTrue from a different class (Assert). The variables “actual” and “expected” are fed into the method. This then gives an answer in the form of true or false. Methods that give something back to the method that called them use “return” to do this. Those methods that don’t return anything are given the prefix “void”. For more information see the links given in the start of the document.

JUnit is a simple framework for writing tests. An in depth documentation can be found at <https://github.com/junit-team/junit4/wiki>. But for now “text fixtures” are the most important element. These are known sets of object/environments against which the tests are run. They give a baseline against which the tests are run and so ensure repeatability. The annotations used are:

@Before=>Annotates the methods that are used before running the test(s). They set up the environment

@After=>These methods will be run after the test execution. They are used to clean up and release resources (like memory) before running the next test (or finishing).

@Test=>Annotates the test that need to be run.

You can see these elements in action in the “google search” exercise on the next page. If there are multiple tests to be run the sequence will be:

@Before🡪@Test 1e function 🡪@After🡪@Before🡪@Test 2e function 🡪@After🡪…

Other functions that are to be called upon don’t need an annotation. These can be called from the respective tests that need them. An example is:

public class LaathetwerkenvOverFuncties {

WebDriver driver;

@Before

public void setDriver() {

System.setProperty("webdriver.gecko.driver", "lib/geckodriver");

driver = new FirefoxDriver();

}

@After

public void cleanup(){

driver.quit();

}

/\*The test itself\*/

@Test

public void theTest(){

//steps for the automation

}

### Using TestNG

Instead of using JUnit, you can also use TestNG. This is an extension of the JUnit framework. TestNG has more annotations to make it more flexible in use. It has as annotations for testbuildup: @Before/AfterClass, @Before/AfterGroup, @Before/AfterSuite, @Before/AfterTest and @Test. Try it first with the JUnit. When this works copy the element as an exercise (so not to lose the working copy) and try to change it towards the TestNG framework.

## Example GoogleSearch:

Here is mine exercise of the “google search”. Try and compare it with your own. Possibly yours is better, but look if you find something you can learn from.

**public class** Doc1Ex1GoogleVersion05

{  
*/\*---------The basics------------------------------------------------\*/* WebDriver **driver**;  
  
 @Before  
 **public void** setDriver() {  
 System.*setProperty*(**"webdriver.gecko.driver"**, **"lib/geckodriver"**);  
 **driver** = **new** FirefoxDriver();  
 }  
  
 @After  
 **public void** cleanup(){  
 System.***out***.println(**"cleaning actions"**);  
 **driver**.quit();  
 }  
  
  
 */\*----------Functions for using in the test----------------------\*/* **public void** googleSearch(String zoeker) {  
 */\*The point of this function is to recieve a search term (zoeker).  
 The function puts the search term in the searchbox.  
 It then sees if the wikitext on the ggogle page matches the one in*

*the Wiki  
 \* \*/  
  
 //zoekactie uitvoeren op google* **driver**.findElement(By.*id*(**"sb\_ifc0"**)).sendKeys(zoeker);

//Above will not work in Chromedriver. Use the following;

//driver.findElement(By.*id*(**"lst-ib"**)).sendKeys(zoeker);

**driver**.findElement(By.*name*(**"btnG"**)).click();  
 *//wait for propper loading* WebDriverWait wait= **new** WebDriverWait(**driver**,2);  
 wait.until(ExpectedConditions

.*visibilityOfElementLocated*(By.*cssSelector*(**"span.st"**)));  
  
 *//Looking at the text* String textwiki = **driver**.findElement(By.*cssSelector*(**"span.st"**))

.getText();

System.***out***.println(textwiki);  
 String InWiki = textwiki.split(**"\\.\\.\\."**)[0];  
 System.***out***.println(InWiki);  
  
 *//Move to the wikipage* **driver**.findElement(By.*partialLinkText*(**"Wikipedia"**)).click();  
 wait.until(ExpectedConditions.*visibilityOfElementLocated*

(By.*xpath*(**"//body/div[3]/div[3]/div[4]/p[1]"**)));

*//See if we find the same text* String TextInWiki = **driver**.findElement(

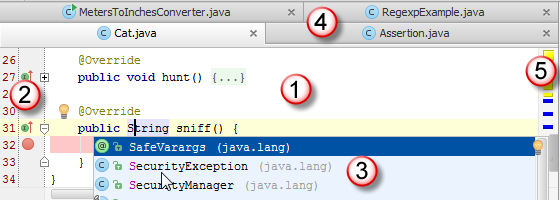
By.*xpath*(**"//body/div[3]/div[3]/div[4]/p[1]"**)).getText();

System.***out***.println(TextInWiki);  
 Assert.*assertTrue*(TextInWiki.contains(InWiki));  
 System.***out***.println(**"assert succesful"**);

*//naar vorige pagina gaan en dan* **driver**.navigate().back();  
 wait.until(ExpectedConditions.*visibilityOfElementLocated*(By.*id*(**"lst-ib"**)));  
 **driver**.findElement(By.*id*(**"lst-ib"**)).clear();  
 }  
  
 */\*----------The test itself--------------------------\*/*  
 @Test  
 **public void** Thetest() {  
 */\*setting starting elements\*/  
 //Setting the implicitwait->click&wait->set it only 1x* **driver**.manage().timeouts().implicitlyWait(5, TimeUnit.***SECONDS***);  
 *//Doing the tests* String[] wikizoek={**"GUI Tests"**,**"Load Tests"**};  
  
 *//Starting page* **driver**.get(**"http://www.google.be"**);  
 **for** (String zoeker:wikizoek) {  
 *//call the function (see previous) and let it do its thing* googleSearch(zoeker);  
 }  
  
 *//did it all work?* System.***out***.println(**"Got to the end, it worked"**);  
  
 }  
}

# Lab02: Debugging

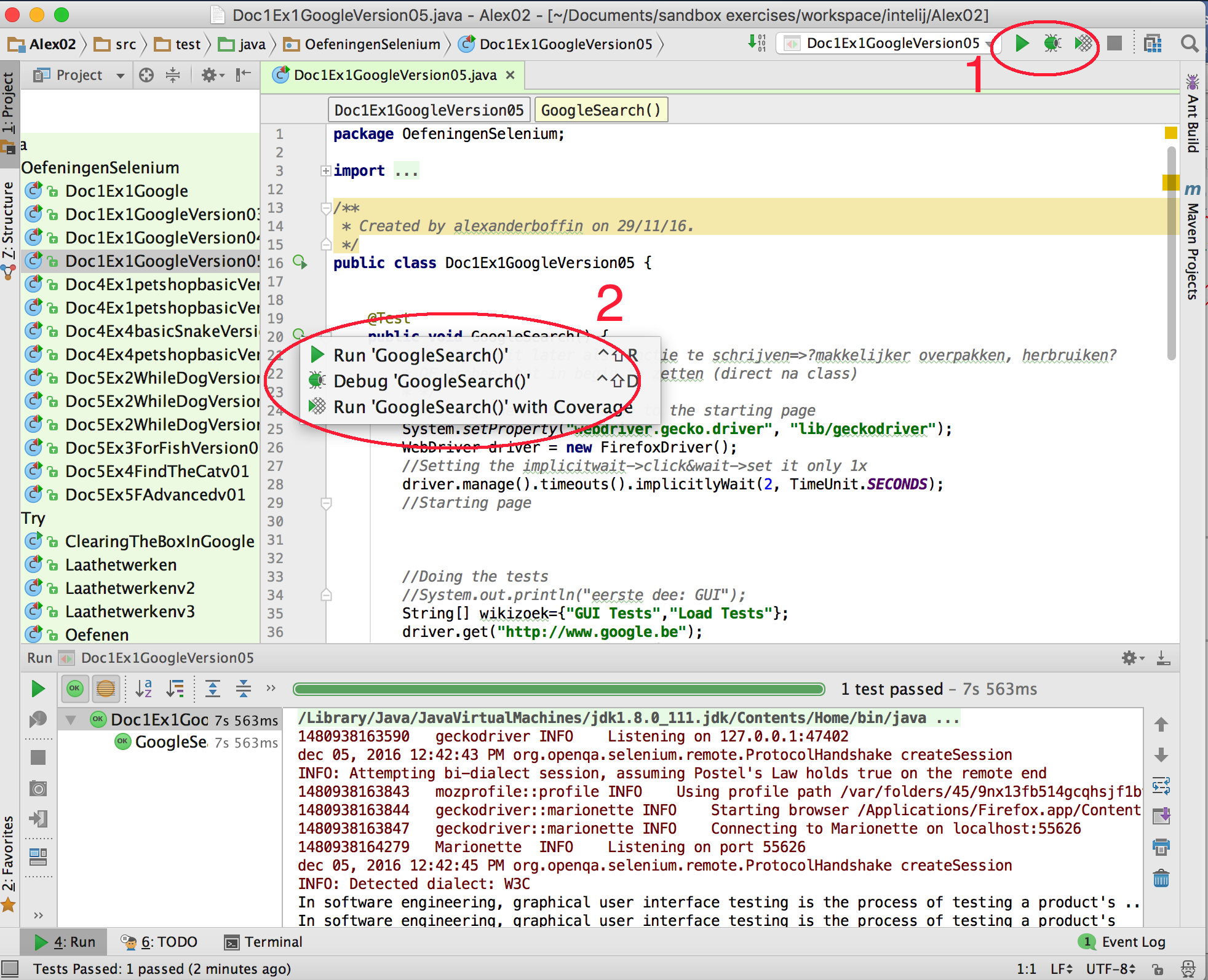
To debug in IntelliJ you must first place a breakpoint in the gutter area (see picture below, number 2). This provides elements to set breakpoints (red dots, can be placed by clicking in the gutter line at the level of the code line where you want the breakpoint to be), scope indicators and the code folding lines that let you hide arbitrary code blocks.



Next you can run through the steps (step by step) in two ways:

1)At the top of the window you select the file you are working on and then click the figure of the bug (is next to the green arrow). If the file itself isn’t in the window, you can try to use 2)

2)By going to the gutter area and selecting the green dot (is at the height of a function or at the height of the class). A little pop up window will be created and you can use Debug ‘javafunction()’.



For a more in-depth information I would recommend looking at :

<https://www.jetbrains.com/help/idea/2016.3/debugging.html>

On this site there is good documentation to see what can be done.

# Lab03: Identifying Elements

For this I would recommend looking back at the file of the Selenium IDE. Firebug doesn’t change with the use of the Selenium WebDriver.

## The XPath Guide:

A handy extension to firebug is FirePath. The extension is an development tool to inspect, edit and generate XPath, CCS and JQuary selectors. This allows to get the relative paths. If you aren’t sure if you have the right element selected with the search string you are using, you can copy and paste it into the FirePath extension. This will either highlight the element it points to ore tell you the element doesn’t exists. The elements of the XPath described below can be visualised with this.

The extension can be can be downloaded at:

<https://addons.mozilla.org/nl/firefox/addon/firepath/>.

Considering XPath is something you’ll be using frequently it is a good idea to get some more information on it. The XPath selector uses the HTML DOM (Document Object Model) structure that can be seen as a tree of objects:



There are two types of XPath as previously mentioned. The first is the “absolute path” that goes through the structure starting from the root element. The search structure used starts then with a single backslash (“/”). For example:

html/body/div[1]/section/div[1]/div/div/div/div[1]/div/div/div/div/div[3]/div[1]/div/h4[1]/b

A more flexible type is the “relative path”. Here the search can start anywhere in the tree structure. It can hold up when small changes are made to the webpage architecture. The search starts here with a double backslash (“//”). The example above can then be set as:

//div/div[3]/div[1]/div/h4[1]/b

Other elements can be used to help select more dynamic elements:

1)The usage of attributes. Here the syntax is:

//tagname[@attribute=’value’]

2)Contains(). This method is used when the attribute is highly dynamic. It uses partial text of the element we’re looking for. The syntax is as follows:

//\*[contains(@attribute=’partial’)]

3)Using OR/AND. Here you can say that the element needs to comply with both or either condition of the attributes. In the “OR” method any one condition needs to be true. The syntax for this is:

//\*[@attribute1=’value1’ OR @attribute2=’value2’]

In the “AND” method both conditions need to be true to find the web element. The syntax for this one is:

//\*[@attribute1=’value1’ AND @attribute2=’value2’]

4)Start-with function. This is to find elements that have an attribute value that changes when the webpage is refreshed or changes dynamically. The syntax here is:

//tagname[starts-with(@attribute=’start’)]

5)Text(). With this elements can be found that have the exact text match as indicated.

//tagname[text()=’texttofind’]

This method can be combined with contains(). The text() becomes here the attribute. This allows you to search the page for any element with the partial text.

//\*[Contains(text()=’partialtext’)]

Remark: Here you might think of redoing the google exercise with this new toy. But watch out: On the Wikipage the text in the first paragraph (what is displayed on the Google) is divided into several partial texts (set it in bold, implementing a weblink,…). So it is more easier to take an element higher in the tree and “remembering the text” so it can be compared to the previous entry. This makes it clear it is always a good idea to see what the best approach is using Firebug as a starting point.

6)The XPath axes methods. This searches different nodes from a particular context node. This is handy for when the elements are highly dynamic, change on refreshing the page or aren’t possible with the aforementioned methods.

a)Following. This will select all elements after the selected node in the tree:

//\*[@attribute='value']//following::tagname

b)Ancestor. This will select all elements that are above the node specified:

//\*[@attribute='value']//ancestor::tagname

c)Child. This will select all direct lower elements of the node.

//\*[@attribute='value']//child::tagname

d)Preceding. This selects all the nodes that are before the current node.

//\*[@attribute='value']//preceding::tagname

e)Following-sibling. This will select the nodes that are at the same hight as the selected node.

//\*[@attribute='value']//following-sibling::tagname

f)Parent. This wil select the node above the selected node.

//\*[@attribute='value']//parent::tagname

g)Self. This will select the node selected.

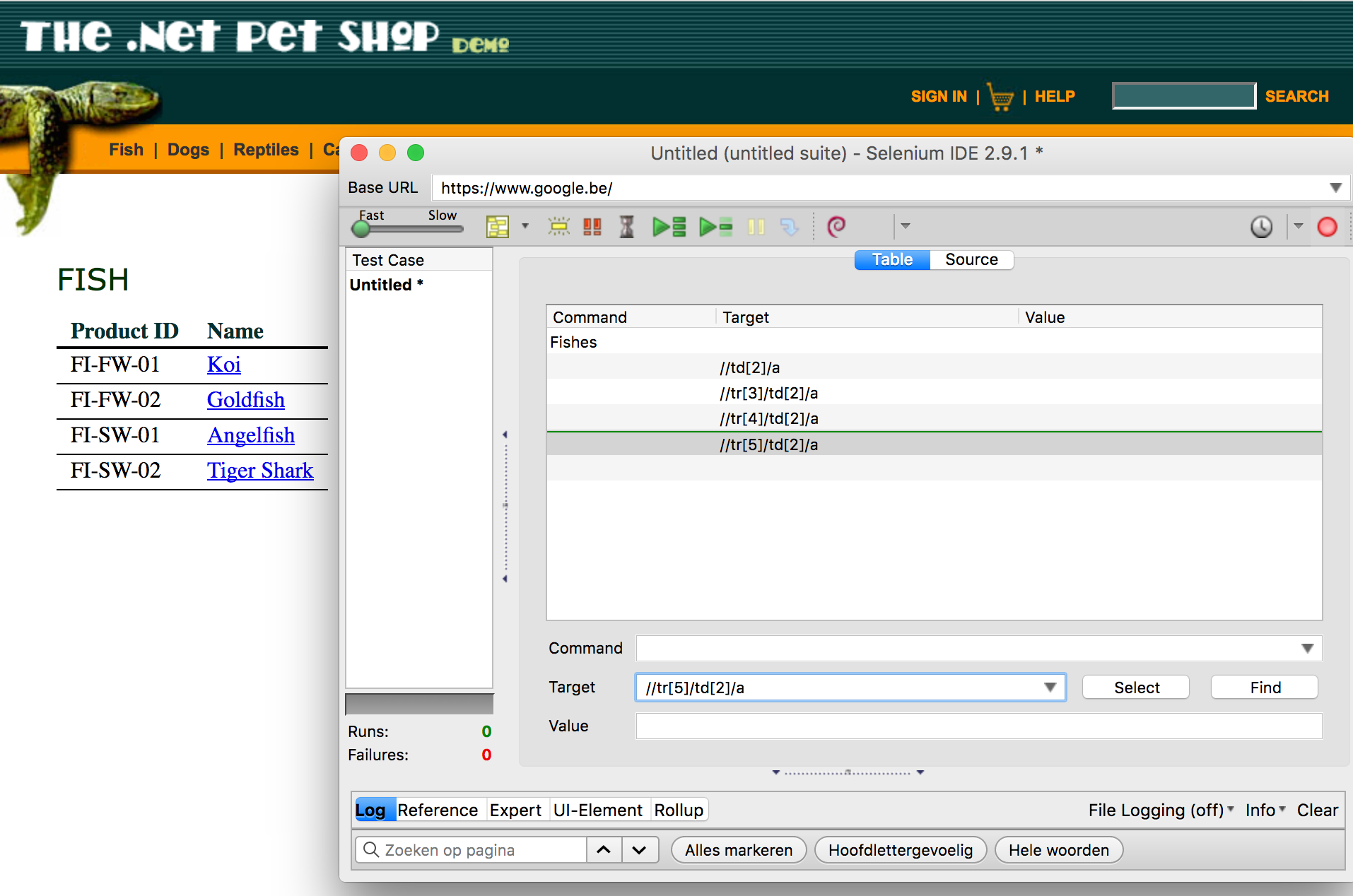
//\*[@attribute='value']//self::tagname

h)Descendant. This will select the nodes that come after the selected node in the tree structure.

//\*[@attribute='value']//descendant::tagname

## Selenium IDE Remark:

Like previously stated it can be helpful to use the select-function of the IDE to see what it would recommend. When needing to select multiple elements on a page and using the Xpath, you can make a list in Selenium IDE. Here you see what they have in common and make a plan to tackle the problem (when working with loops).



# Lab04: Selenium Syntax

Most of these elements have been seen in Lab01 (see back). Some new elements that you’ll need are introduced:

## A)Assert The Title

To get the title and store it you use .getTitle():

String thetitle=driver.getTitle();

If you want to know if it is the expected text, you use the assertEquals or assertTrue statements mentioned in Lab 01.

Assert.*assertEquals*(titleofpage,theexpectedtitle);

Assert.*assertTrue*(titleofpage.contains(theexpectedtitle));

This can also be done in one step:

Assert.*assertEquals*(driver.getTitle(),”theexpectedtitle”);

## B)Operators

Concatenate 2 strings:

System.***out***.println(**”Animal of the list”**+**"-link has been found"**);

To use mathematical operators:

+ Additive operator (also used for String concatenation)

- Subtraction operator

\* Multiplication operator

/ Division operator

% Remainder operator

(!)Watch out with float and double characters when doing precise calculations. If you need to have exact calculations in the future use .math.BigDecimal.

To round off you can use Math.round().

Spiting a string: Use .split(“split marker”)

String str=”0967-97766”;

String[] partstring=str.split(“-”);

System.out.println(partstring[0]);=>gives 0967

(!)To escape special symbols you can use \\ to turn it into a Java String Literal. The first “\” will set the special character into a meta-character. The second “\” will escape the meaning of the “\” as an escape to a string literal.

Making substrings

.substring(int X)

=> will make a new string beginning at character with index X

.substring(int X,int Y)

=> will make a new string beginning index X and ending at index Y-1

(!)start counting at 0

(!)=>the character with index Y will not be included in the substring. You get [X, Y[ not [X, Y]. So if you need until the nth character then use .substring(startcharacter, n+1).

Upper & lowercase

To set a string to lowercase lettering: stringname.toLowercase()

To set a string to uppercase lettering: stringname.toUppercase()

Replacing elements in a string

The most common way to replace a character is to use;

string.replace(‘character1’,’character2’)

To remove all whitespaces you can use the following;

string.replaceAll(**"\\s+"**, **""**)

## C)EXTRA: Handy, But Not Essential Tricks

Going back to the previous webpage

driver.navigate().back()

Selecting the URL of the current page:

driver.getCurrentUrl()

To select an element in a dropdowm window(see figure):

Select select = **new** Select(driver.findElement(By.*id*(**"addr\_listState"**)));  
select.selectByValue(**"New York"**);

You can also use .selectByVisualText or .selectByIndex for this task.



# Lab05: Conditions And Loops

This is just a repeat for those that have taken one of the links to the tutorials in the start of the book. Still look it over just as a reminder.

## A)While Loops

There are two modes to employ the while loop:

Option 1:

while (conditional){

….statements

}

+>Will test the condition first before going to the statements between the brackets.

Option 2:

do{

….statements

}while (conditions);

=>Here the statements between the brackets are done before the condition is tested. This is mostly used if you need the loop to run at least once. Which one you use will mostly depend on the situation at hand and also on your own preference.

## B)The For Loop

The appearance here is like the one in the Selenium IDE:

for(initializer; end condition; increment){

statements

}

example:

for (int i=0; i=<3;i++){

System.out.println(i);

}

EXTRA1: foreach loop. This can be used to iterate over an array.

Int[] TheList={2,3,4,1,2};

for (int elementOfList: TheList){

//Will go over the different elements in the array

System.out.println(elementsOfList);

}

EXTRA2:There is a way to skip an iteration or to exit the loop completely

To stop the loop and go to the next statement outside the loop:

break;

To stop the current iteration and go to the next one:

continue:

=>These two statements can be used to modify the behavior of the loop to get more control on what the loop does.

## C)if, else if And else

The form is:

if(condition1){

Statements if condition 1 is true

}else if (condition2){

Statements executed if condition 1false, but condition 2 true

}else{

These statements will be executed if all of the previous conditions are false

}

Depending on the conditions the if-statement can be used alone, in combination with the else-statement or there can be several else if-statements present. The conditions will be checked one by one until one is found true (or all statements are exhausted or the else statement is reached).

# Lab 06: Data Driven Testing

Before starting with going over the files, I would like to recommend the text editor Komodo. This handy thing can easily make all the different text types for you and is very easy to use.

This editor can be downloaded on <https://komodo-edit.nl.softonic.com/mac>



## A)CSV Files

THE FILE USED

Fish,Koi,Goldfish,Angelfish,Tiger Shark

Dogs,Bulldog,Chihuahua,Dalmation,Poodle

Reptiles,Iguana,Rattlesnake,,

Cats,Persian,Manx,,

Birds,Amazon Parrot,Finch,,

BEFORE STARTING=>need maven file extension. Add the following elements:

<**dependency**>  
 <**groupId**>com.opencsv</**groupId**>  
 <**artifactId**>opencsv</**artifactId**>  
 <**version**>3.8</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.apache.commons</**groupId**>  
 <**artifactId**>commons-csv</**artifactId**>  
 <**version**>1.3</**version**>  
</**dependency**>

Check the values in red with the latest version. These can be found at the MVN repository <https://mvnrepository.com/> .

USING IT IN THE PROGRAM

//first you’ll need to read in the file itself

CSVReader reader = **new** CSVReader(

**new** FileReader(**"/Path to file/file.csv"**));

//Now you can load its content into a list

List<String[]> li=reader.readAll();

//With this you can iterate over the values of the file

**while** (i1.hasNext()){  
 String[] str=i1.next();  
 System.***out***.println(**"values are"**);  
 **for** (**int** i=0;i<str.**length**;i++){  
 System.***out***.print(**" "**+str[i]);  
 }  
 System.***out***.println(**""**);  
}

THE OUTPUT:

values are

Fish Koi Goldfish Angelfish Tiger Shark

values are

Dogs Bulldog Chihuahua Dalmation Poodle

values are

Reptiles Iguana Rattlesnake

values are

Cats Persian Manx

values are

Birds Amazon Parrot Finch

EXTRA(!)

In the Selenium IDE it was possible just to read the value of a particular row, column. At the time I’m typing this I haven’t found a function that can do this as such. But I did try to create something similar into a function. This can then be placed before the main function and called upon whenever needed.

**public static** String[] CSVLine(String FileCSV, **int** TheLine) **throws** IOException {  
 *//Trying to create a recallable function to satisfy my personal need* CSVReader reader = **new** CSVReader(**new** FileReader(FileCSV));

*//Creating an empty array*  
 String[] csvCELL = **new** String[0];  
 **for** (**int** ink=1;ink<=TheLine; ink++){  
 csvCELL = reader.readNext();  
  
 }  
 **return** csvCELL;  
  
 }  
  
**public static void** main(String[] args) **throws** IOException {  
 */the function I developed*

String[] WhatWeNeed=*CSVLine*(**"Path to file“**,3);

*//Iterating over the given array*  
 **for**(String el:WhatWeNeed) {  
 System.***out***.println(el+**" "**);  
 }

*//Selecting a particular elemen/column from the selected row*  
 System.***out***.println(**"is this right? :"** +WhatWeNeed[1]);  
}

THE OUTPUT:

Reptiles

Iguana

Rattlesnake

is this right? :Iguana

## B)XML Files

No need to add a Maven element for this one.

THE FILE USED

<testdata>

<vars animal="Fish" subcategory="Saltwater,Freshwater"/>

<vars animal="Dogs" subcategory="Poodle,Greyhounds"/>

<vars animal="Reptiles" subcategory="Iguanas,Snakes,Turtles"/>

<vars animal="Cats" subcategory="Manx,Persian"/>

<vars animal="Birds" subcategory="Eclectus,African Greys,Macaws"/>

</testdata>

USING IT IN THE PROGRAM

*//initialisation reading xml file*File inputFile = **new** File(**"/path/filexml"**);  
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.*newInstance*();  
DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();  
Document doc = dBuilder.parse(inputFile);  
*//An extra step to reduce redundancies and easier tob e read*

doc.getDocumentElement().normalize();

*//Get the root of the document& use .getNodeName() to convert it to a string*String root = doc.getDocumentElement().getNodeName();  
System.***out***.println(root);

*//Making a list of the different nodes(here the <vars />*NodeList Varslist=doc.getElementsByTagName(**"vars"**);  
 *//go tot he node that has the needed information*Node Varsregel=Varslist.item(1);  
System.***out***.println(Varsregel.getNodeName());  
  
*//getting to the attributes we want!!!!!!!*Element TheAtribute=(Element) Varsregel;  
System.***out***.println(TheAtribute.getAttribute(**"subcategory"**));

*//trying to make it shorter*Element TheAtribute2=(Element) doc.getElementsByTagName(**"vars"**).item(1);  
  
System.***out***.println(**"trying something: "**+TheAtribute2.getAttribute(**"subcategory"**));  
  
*//Further condensation:*String TheAtribute3=((Element) doc.getElementsByTagName(**"vars"**).item(1)) .getAttribute(**"subcategory"**);  
  
System.***out***.println(**"Further concentrated gives :"**+TheAtribute3);

THE OUTPUT:

testdata

vars

Poodle,Greyhounds

trying something: Poodle,Greyhounds

Further concentrated gives :Poodle,Greyhounds

## C)JSON FILE

There are several possibilities of API’s that can be used. For a quick overview you can look at <http://tutorials.jenkov.com/java-json/index.html>. But here we use the JSON-SIMPLE API.

Side information:

You can later experiment with the different types and see what you like better. Watch out on different forums. The different types have other possible commands and it can get frustrating if you follow an example only to find the command is of an API that is not the one you are using. Also if using multiple API’s watch out that the JSONObject and other similar elements are of the correct API (or that they are compatible). A org.json.simple can’t be used first on something and later be used by an element of org.json.

(Of course this can change if a new update arrives after I wrote this. That is why it is important to do experiments. But for now use the JSON.simple to get a grip on the basics and experiment later on if you have some spare time or so).

BEFORE STARTING=>need maven file extension. Add the following elements:

<**dependency**>  
 <**groupId**>com.googlecode.json-simple</**groupId**>  
 <**artifactId**>json-simple</**artifactId**>  
 <**version**>1.1.1</**version**>  
</**dependency**>

Again make sure that you use the latest version. This can be checked like previously stated at the MVN repository <https://mvnrepository.com/>

THE FILE USED

{"Name":"The Petshop",

"AnimalTypes":["Fish","Dogs","Reptiles","Cats","Birds"],

"TheHomepage": [

{ "animal": "FISH", "subcategory": "Saltwater,Freshwater" }

,{ "animal": "DOGS", "subcategory": "Poodle,Greyhounds" }

,{ "animal": "REPTILES", "subcategory": "Iguanas,Snakes,Turtles" }

,{ "animal": "CATS", "subcategory": "Manx,Persian" }

,{ "animal": "BIRDS", "subcategory": "Eclectus,African Greys,Macaws" }

]

}

USING IT IN THE PROGRAM

*//read the file itself*org.json.simple.parser.JSONParser parser=**new** org.json.simple.parser.JSONParser();  
  
JSONObject obj = (JSONObject) parser.parse(**new** FileReader(**"/Users/alexanderboffin/Documents/sandbox exercises/selenium/seleniumbasicexercises/data driven testing/doc6ex6d.json"**));  
  
*//Getting one of the variables (outside an array []*String name= (String) obj.get(**"Name"**);  
  
System.***out***.println(name);  
  
*//Getting an element from a normal array*org.json.simple.JSONArray arr= (org.json.simple.JSONArray) obj.get(**"AnimalTypes"**);  
System.***out***.println((String) arr.get(1));  
  
*//getting elements out an arry with selectables*org.json.simple.JSONArray arrSP= (org.json.simple.JSONArray) obj.get(**"TheHomepage"**);  
  
JSONObject objOfArray = (JSONObject) arrSP.get(1);*//setting found element to new object*String nameOfArrayElement=(String) objOfArray.get(**"subcategory"**);  
System.***out***.println(nameOfArrayElement);

THE OUTPUT:

The Petshop

Dogs

Poodle,Greyhounds

# Lab 08: Extras For WebDriver

## A)Using Different Web Browsers.

In the set-up documentation we stated to download the geckodriver and then place it in the “lib” folder. This can also be done for the other drivers.

The ChromeDriver

Add to the pomfile:

<**dependency**>  
 <**groupId**>org.seleniumhq.selenium</**groupId**>  
 <**artifactId**>selenium-chrome-driver</**artifactId**>  
 <**version**>3.0.1</**version**>  
</**dependency**>

Go to <https://sites.google.com/a/chromium.org/chromedriver/downloads>

Here you can download the latest release. Just put it into a folder you can find back.

Unzip the file and then copy it into the same folder as the geckodriver.

The only thing you now need to do is replace the statements with the geckodriver into;

System.*setProperty*(**"webdriver.chrome.driver"**, **"lib/chromedriver"**);  
WebDriver driver = **new** ChromeDriver();

Normally you can now do the same tests of the firefoxdriver with the chrome driver. To check if everything is alright try to open and close the chrome window on a website. If a test in Firefox passed and not in chrome, look with the debugger where it falters. It can be that some findElements can’t be found properly. Just change these (The elements most vulnerable are the text fields to send text to and some text elements). If there is a problem, try to use the selenium IDE to see what it suggests and compare it to what you use. An example is the google exercise previously. This will work in Firefox, but fail in chrome (because off **”sb\_ifc0”** which isn’t recognized by Chome as the correct place to send text to). But if you use **”lst-ib”** the chromedriver functions perfectly.

It might also be a good idea to increase the wait times (with me chrome was what slower than the geckodriver).

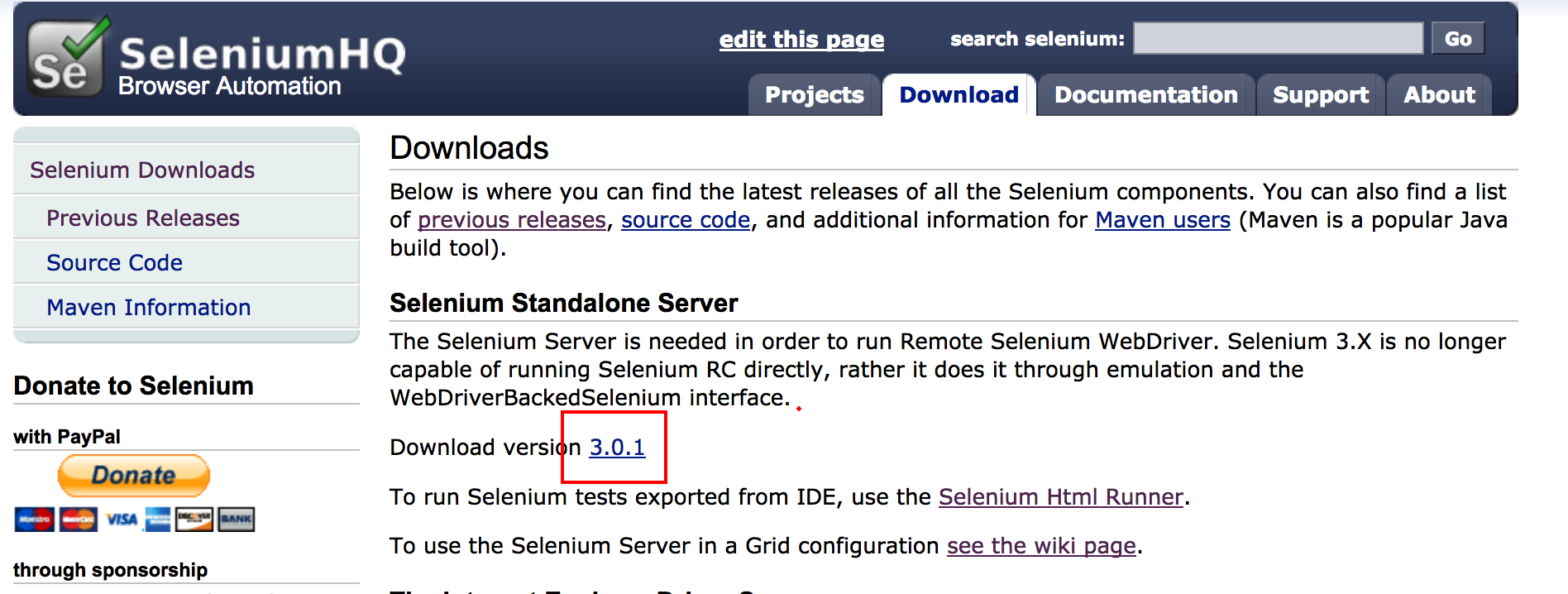
**Safaridriver: still need to research it myself**

## B)Selenium Grid (Doing Parallel Testing

Imagine having 100 tests to run. If you can split those tests over 4 virtual machines, you are done in approximately ¼ the time as if you do them sequentially. The Selenium Grid gives the possibility to do tests in parallel. Also it offers another way to implement different browsers without having to install the drivers (we will see this later on in the text).

### Setting Up The Hubs And Nodes

**Step 1)** getting the latest version of the selenium standalone server. This can be found at <http://www.seleniumhq.org/download/>



Install this in a folder you can locate on your computer.

**Step2)** Now that we have the server, we need to make a Hub on our computer. This hub can be seen as a central command center that will trigger the tests. Navigate on your computer to the Programs folder->Utilities folder->”Terminal”. Click on it



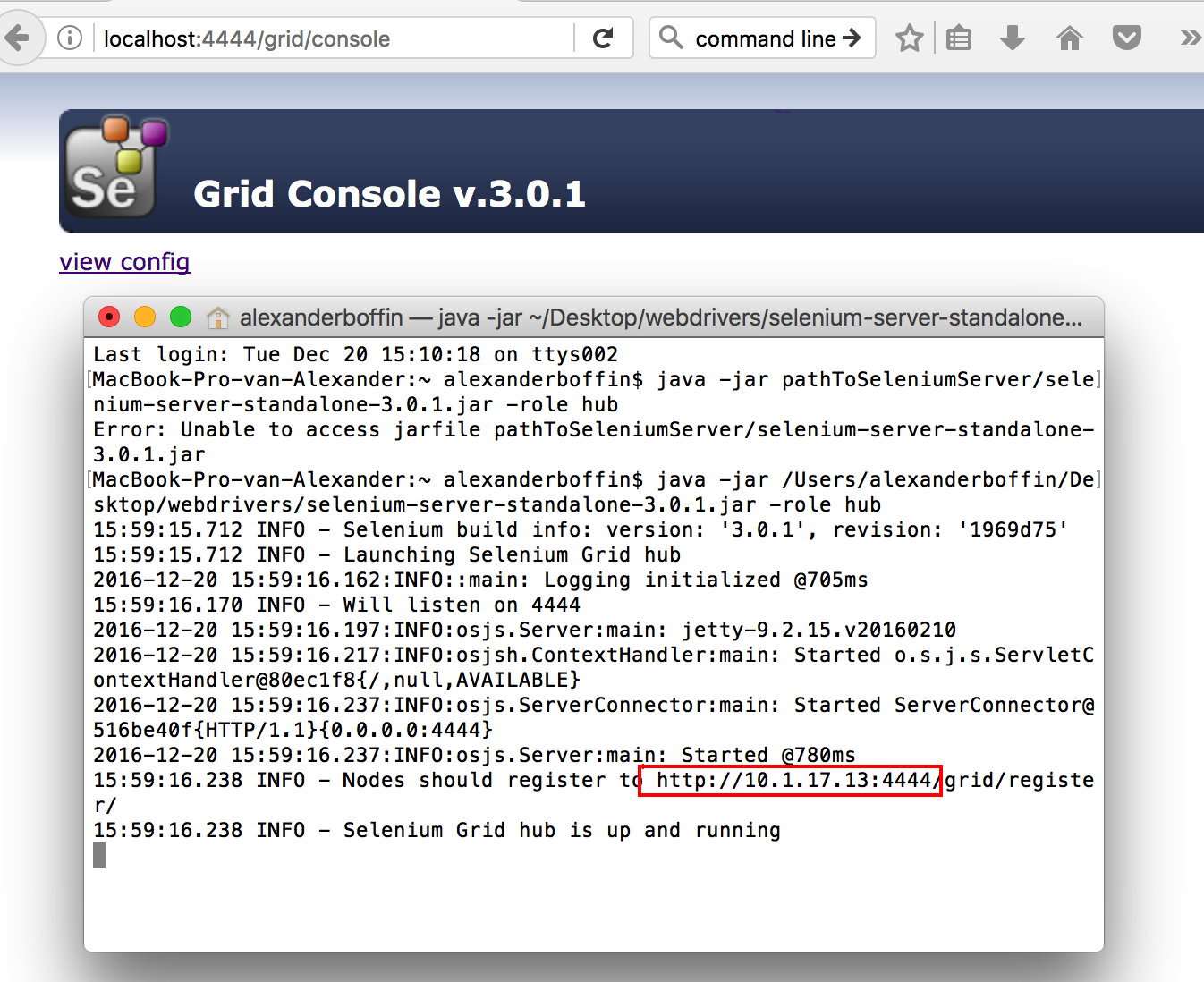
Intermezzo:

Before there were GUI (Graphic User Interfaces) there were CLI (Command Line Interfaces) to move between folders, move objects or anything else. It is a very handy thing to invest some time in later, but for now just follow the following steps.

A window will open. In this you can type:

java -jar pathToSeleniumServer/selenium-server-standalone-3.0.1.jar -role hub

This will create a hub. You can check if you are successful by using a browser and go to <http://localhost:4444/grid/console>. Here you’ll find the following:



Note the localhost number (in red before the :4444). This will be used in the following step (creating a node).

**Step3**) Now we are going to create a node. This can be viewed as setting up the browser(s) that will run the tests. Open a new command line terminal and input the following (Input the local host that was created):

java -jar pathToSeleniumServer/selenium-server-standalone-3.0.1.jar -role webdriver -hub http://localhost:4444/grid/register -port 5588

Refresh the <http://localhost:4444/grid/console>. Here the following should be seen:



Remarks: If “–port 5588” isn’t put into the command line the node will be made at port 5555 (you can select the ports from 5555 to 5590). To make other nodes, just change the number here.

**Step4)** Because we want to run several tests in parallel you need an equal number of nodes as the tests run in parallel. To do this run step3 as many times as necessary with different port numbers.

### Method 1: To Make Tests Run In Parallel Using JUnit

You can use several ways to run the tests in parallel and two are used here (you will need to have set up 2 nodes for both tutorials). Which one you chose is up to you and your needs at the moment. Keep in mind that there are more ways to give a cat a bath and the same goes here. There are probably more ways out there if you look.

The first uses the JUnit framework. An element is added to the pom-file for running the tests in parallel or sequentially. This is the simplest method to use. Each test can also be done separate by running the test like before.

**Step1a)** Previously we used the geckodriver and Chromedriver as WebDriver, now we are going to use RemoteWebDriver. Unther the Class you put:

RemoteWebDriver **driver**;

To use the driver now you add

DesiredCapabilities capability = DesiredCapabilities.*firefox*();  
capability.setBrowserName(**"firefox"**);

**driver** = **new** RemoteWebDriver(

**new** URL(**"http://localhost:4444/wd/hub"**), capability);

This replaces:

System.*setProperty*(**"webdriver.gecko.driver"**, **"lib/geckodriver"**);  
**driver** = **new** FirefoxDriver();

Another class can be made to execute the test by changing the elements of the DesiredCapabilities to:

DesiredCapabilities capability = DesiredCapabilities.*chrome*();  
capability.setBrowserName(**"chrome"**);

**Step1b)** You can check if the tests works by running them like before (select the test to run on the top right, with the green triangle or you select the green orb at the height of the class).

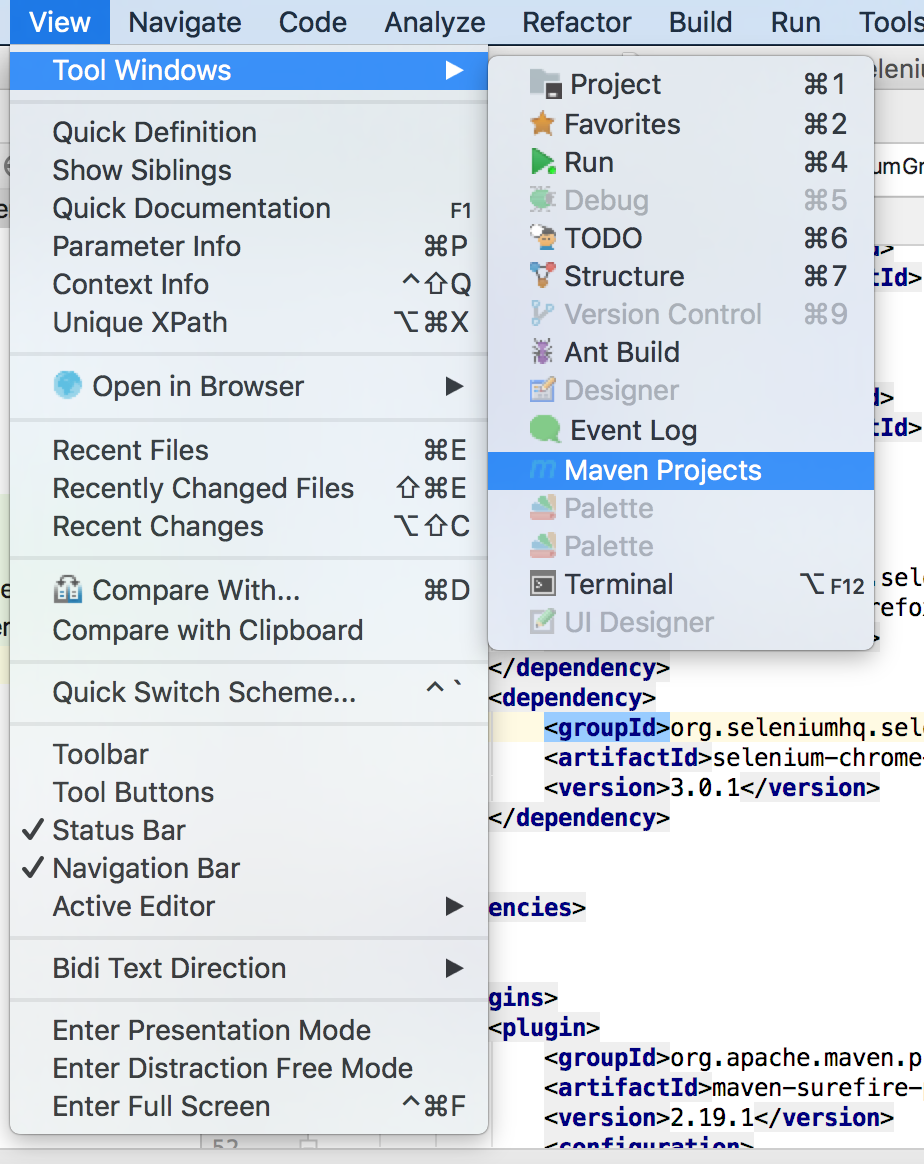
**Step2a)** The tests can be run sequentially by going to the pom-file and adding the following outside the <dependencies> (set it after to be sure):

<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-surefire-plugin</**artifactId**>  
 <**version**>2.19.1</**version**>  
 <**configuration**>  
 <**includes**>  
 <**include**>pakage.nameClassToTest1</**include**>  
 <**include**>pakage.nameClassToTest2</**include**>  
 </**includes**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

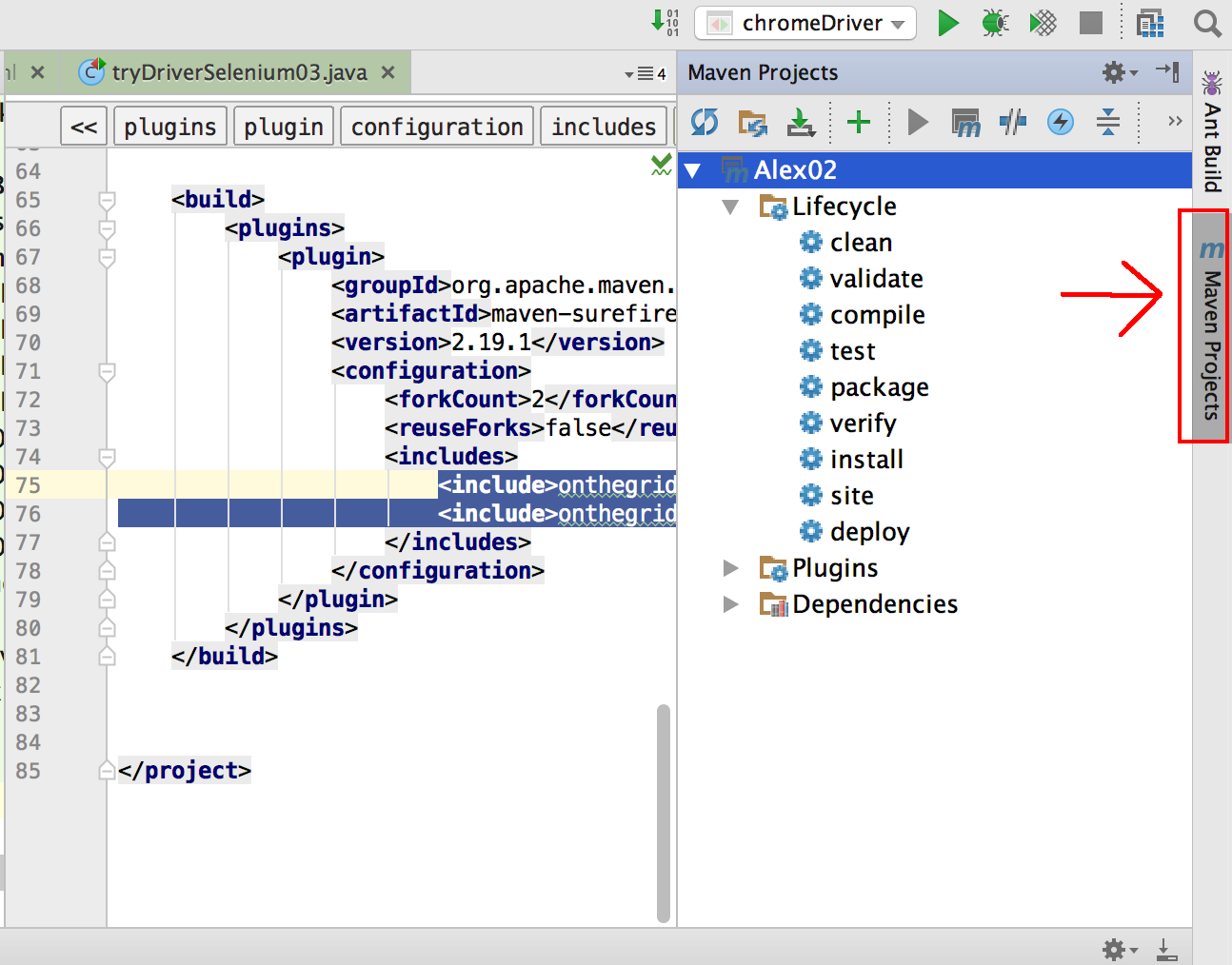
Don’t forget to see if the version is the most recent.

To refer towards the files (Classes) that need to be tested you set first the package name and then the name of the class that needs to be tested after the “.”. For more info on how to include tests you can see <http://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html>.

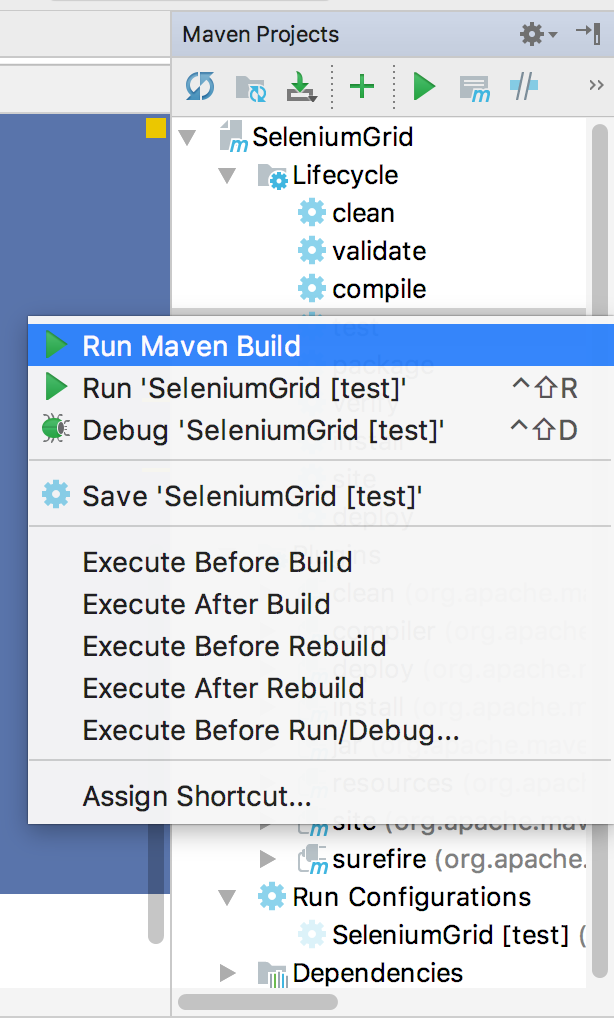
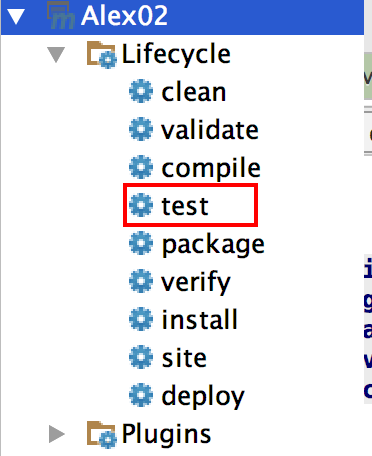
To run this, you first need to open the “Maven Projects” window. This can be done by selecting View->Tool Window->maven Projects. This will open the window.



Another way to open the window is by clicking on “Maven projects” that is displayed right.



Here you can left click on “test”. In the new window you can select the “Run maven build”. This will run the selected test in maven sequentially



Step2b) For running the tests sequentially you only need to change the pom-file as seen here:

<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-surefire-plugin</**artifactId**>  
 <**version**>2.19.1</**version**>  
 <**configuration**>  
 <**forkCount**>2</**forkCount**>  
 <**reuseForks**>false</**reuseForks**>  
 <**includes**>  
 <**include**>pakage.nameClassToTest1</**include**>

<**include**>pakage.nameClassToTest2</**include**>  
 </**includes**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

To run the test here you do the same as in step 2a)

### Method2: Run Tests In Parallel Using TestNG

Here a different framework is used: TestNG (see separate document over the differences between this one and JUnit). TestNG lets us use parameters we can set in an xml-file.

**Step1)** Make the Class to be tested. Here we need to change the annotations to those of the TestNG (@BeforeTest, @AfterTest and @Test). In the previous we needed to make two files for doing the test in two browsers. Making use of the fact that TestNG lets you use parameters that can be set in the xml-file (see further on), you only have to make 1 file.

This is done by setting the @Parameters(“”) above the method we need to use the parameters (when setting the Remotedriver in the example below). This can then be fed into the method (see back Doc 1 part J). Next the different Remotebrowsers are encapsulated in if-if else functions. Depended on the set parameter in the xml-file a different driver is used. The else element here is used to see if the driver is wrongly set in the xml-file.

To let the test run on its own you use @Optional(“”) to feed it a standard value. If you don’t use the XML-file this value will be used instead. Now you can run it like any other test and debug.

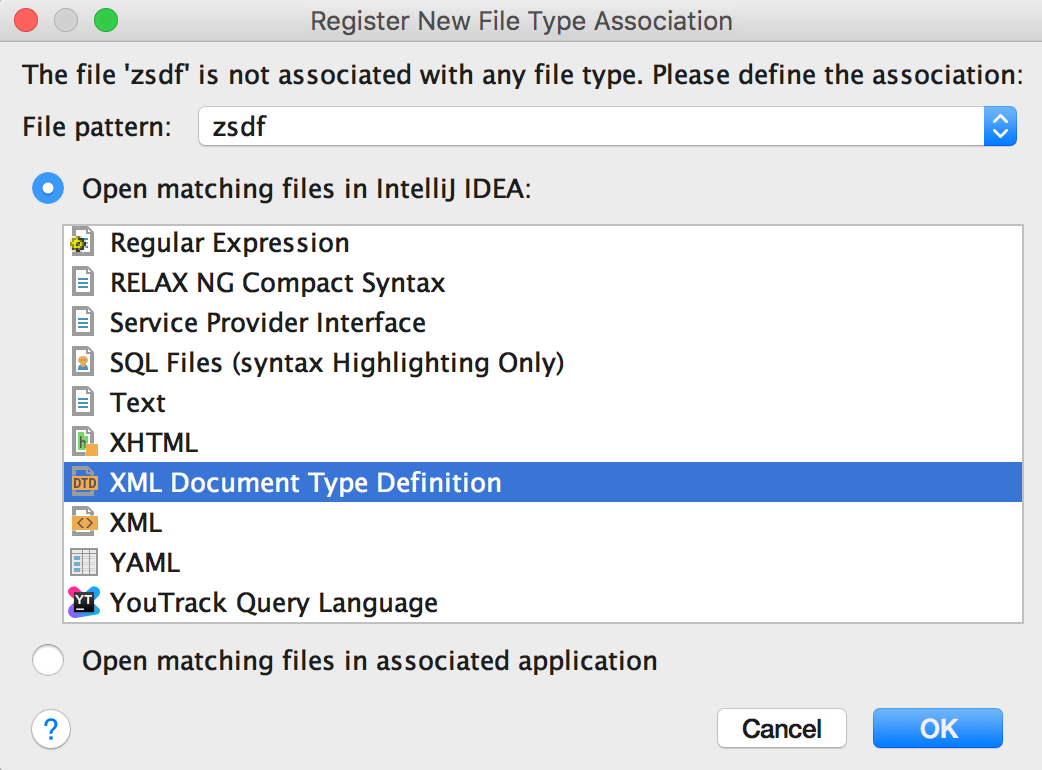
@Parameters(@Optional(**"firefox"**)**"browser"**)  
@BeforeTest  
**public void** launchDriver(String browser) **throws** MalformedURLException {  
 **if** (browser.equalsIgnoreCase(**"firefox"**)) {  
 System.***out***.println(**" Executing on FireFox"**);  
  
 DesiredCapabilities capability = DesiredCapabilities.*firefox*();  
 capability.setBrowserName(**"firefox"**);  
  
 *//setting the driver* **driver** = **new** RemoteWebDriver(**new** URL(**"http://localhost:4444/wd/hub"**), capability);  
 System.***out***.println(**"Firefox driver set"**);

}**else if**(browser.equalsIgnoreCase(**"chrome"**)){  
 System.***out***.println(**" Executing on Chrome"**);  
  
 DesiredCapabilities capability = DesiredCapabilities.*chrome*();  
 capability.setBrowserName(**"chrome"**);  
  
 **driver** = **new** RemoteWebDriver(**new** URL(**"http://localhost:4444/wd/hub"**), capability);  
 System.***out***.println(**"Chrome driver set"**);  
  
 }**else**{  
 **throw new** IllegalArgumentException(**"The Browser Type is**

**Undefined"**);  
 }  
  
  
}

Step 2) Now we are going to make the XML-file to use. First click left on the package where you are working in. Select new->File. Next you can select to use a file with XML-properties.

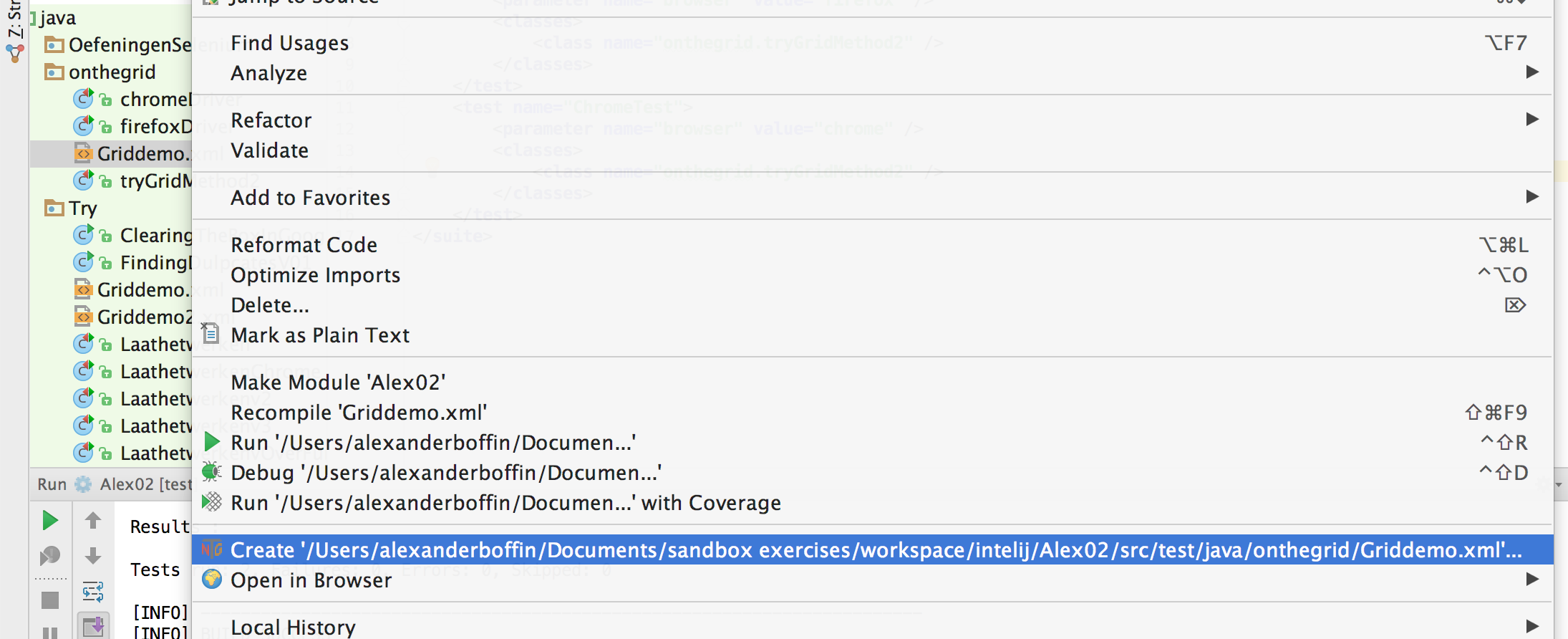


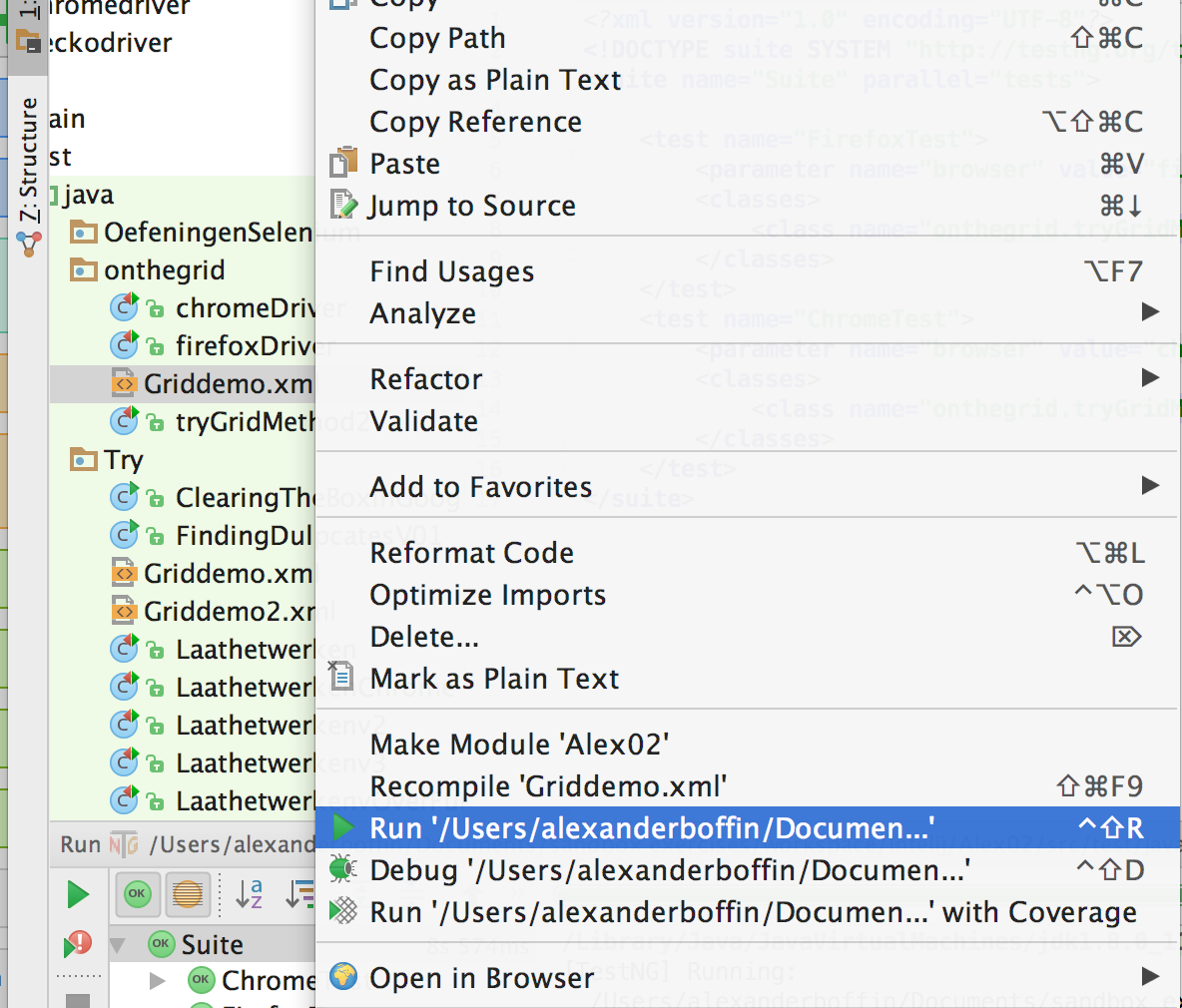


In the file you can now put the following:

*<?***xml version="1.0" encoding="UTF-8"***?>***<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd"*>***<**suite name="Suite" parallel="tests"**>  
  
 <**test name="FirefoxTest"**>  
 <**parameter name="browser" value="firefox"** />  
 <**classes**>  
 <**class name="**pakage.nameClassToTest1**"** />  
 </**classes**>  
 </**test**>  
 <**test name="ChromeTest"**>  
 <**parameter name="browser" value="chrome"** />  
 <**classes**>  
 <**class name="**pakage.nameClassToTest1**"** />  
 </**classes**>  
 </**test**>  
</**suite**>

**Step3)** Now the XML-file is almost ready for doing the same test in parallel over two different browsers. Click left on the made xml-file and then on “create” just under run. In the window that appears you select yes. When you have done this you can run the test by again clicking the XML-file and now selecting “Run”





Step3b)If you want to run the test sequentially, you only need to delete in the file:

**”parallel="tests"**

When you run the test now, the tests will be done sequentially.

# Lab 09: Advanced Stuff

## iFrames

The term iFrame stands for inline frame. These are HTML elements that are imbedded into another HTML document (That again can be inside another webpage). Example you can find regularly are advertisements, the usage of applications imbedded in websites and YouTube videos that are placed inside a webpage.

These frames are sometimes frowned upon because search engines aren’t always capable of relaying the contends within. Also sites using iFrames will display multiple URL’s. But as a tester it is important to know about these elements. For example, go to the website: <http://www.guru99.com/selenium-tutorial.html>. There are several advertisements. If you try to use the Selenium IDE, you’ll find that it can’t be selected with the “select button”. When inspecting the element using Firebug it is apparent that the element is inside an iframe (that itself is inside an iframe). So to test these elements you need a different approach.

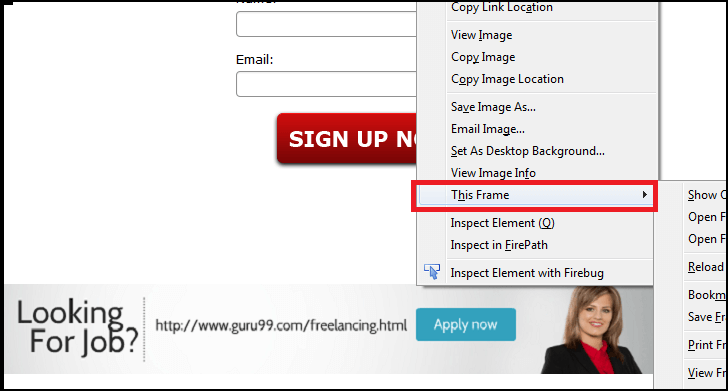


### See If There Are iFrames On The Page:

There are several ways of finding out if there are iFrames on the webpage.

1)Use Firebug to do a search for “iframes”. This will let you cycle through the different iframes. (<iframes isn’t searchable)

2)To see if the element is an iFrame, you rightclick on it. If it is a iFrame there should be an “This Frame option” (if in duch language: “Dit deelvenster”).

(in Engish OS)

(For on duch computer)

3)Using selenium to identify the number of iFrames on the page. This is done using:

driver.findElements(By.tagName("iframe")).size();

### Switching To The iFrame Using WebDriver

1) Using the index (count starts at 0)

driver.swichTo().frame(nummer);

2)Using the Name or ID of the iFrame

driver.swichTo().frame(“nameorid”);

3)The possibility exists to capture the WebElement earlier in the Class (using one of the other ways seen: can’t navigate to element inside iFrame, but can navigate to the frame itself). This is done by:

driver.swichTo().frame(WebElement);

### Switch Back To A Previous Frame

To move back to the parent frame, you use:

driver.swichTo().parentFrame();

If you need to go back swiftly to the main frame:

driver.swichTo().defaultContent();

### The Concept Of Nested Frames:

The example given in the start is a iFrame within another iFrame. In this case you’ll need to navigate through both frame to get to the needed element. So you need to do:

1)switch to the outer frame (see previous)

2)Switch to the wanted inner frame

3)Repeat step2 if necessary

4)You can now select the element needed.

Like you can see iFrames can quickly complicate testing if there are a lot of nested iFrames. This becomes even more complicated when the page being tested is highly dynamic and in flux.

## Parameterization With TestNG=>Data Driven Testing

If we want to make sure that the software is capable of operating using different sets of data, we need to be able to feed this to the system under test. This is part of the TDD (Test Driven Development). We do this using parameterization. Here this will be explained with TestNG There are 2Types of parameterisations:

### (1)Using The Parameter Annotation And The xml-file

a) Use the @Parameters annotation above the method to be tested. Here you add the parameters that need to be taken from the xml file. Here is an example:

@Test

**@Parameters({"start","addition"})**

public static void exampleFunctionToTest(

**@Optional("1")** int startnumber, **@Optional("1")** int addition) {

System.out.println("adding "+startnumber+" + "+addition+"

= "+(startnumber+addition));

}

}

Remark: The **@Optional("1")** allows you to place values that will be given to the method in the absence of other drivers. This allows you to see if the method works initially as planed (can be run like usual then).

b) now we make the XML-file to use:

<?xml version="1.0" encoding="UTF-8"?>

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

<suite name="TestSuite" thread-count="1" >

<parameter name="start" value="98" />

<parameter name="addition" value="2" />

<test name="testGuru">

<parameter name="addition" value="3" />

<classes>

<class name="tryfurtherselenium.tryingParameterization">

</class>

</classes>

</test>

<test name="test2">

<classes>

<class name="tryfurtherselenium.tryingParameterization">

</class>

</classes>

</test>

</suite>

Remark1: Above there are two “<parameter name="addition"”. In this case the parameter of the class gets precedent over the other one (of the test suite). Testscases without the parameter will use the one of the testsuit.

Remark2: Like you can see above, the name of the parameter in theXML-file must be the same as the one in the class given in the @Parameters annotation. But this doesn’t need to be the same as the attributes in the method itself (just need the right place).

Remark3: If you want to use multiple values of the same parameter, you can eather make a long and difficult to overlook XML-file. Or you can use the DataProvider discussed below.

### (2a)Parameters Using The DataProvider (Default)

This method is used to create large datasets in a single execution. To use, will need to create a method that can give values and annotate it with @DataProvider. At the method to be tested we add dataProvider=”nameofprovider”. This will tell it which provider to use.

You can leave out the “(name=”nameofprovider”)” at the @DataProvider. But then you need to set the name of the method used as a provider to the @Test(dataProvider="methodusedasprovider")

An example:

**@Test(dataProvider="countProvider")**

public static void exampleFunctionToTest(

@Optional("1") int startnumber, @Optional("1") int addition) {

System.out.println("adding "+startnumber+" + "+addition+"

= "+(startnumber+addition));

}

**@DataProvider(name="countProvider")**

**public Object[][] getDataFromDataprovider(){**

**return new Object[][] {**

**{ 1, 1 },**

**{ 2, 5 },**

**{ 1000, 234 }**

**};**

Remark1: If we want to set the DataProvider into a different Classfile. Then we need to:

-set the class parameter of the provider to static

-Set at the method to be tested:

@Test(dataProvider="SearchProvider",

dataProviderClass=pakagename.classname)

Remark2: There are 2 types of parameteers supported by DataProvider. The two types are discussed in (2B) and (2C)

### (2B) Parameters Using The DataProvider (Method)

This is used when you want to use one DataProvider method that can behave differently depended on the method to be tested. This is specified using the “java.lang.reflect.Method”. This gives information about a method and permits access to it. So you can use conditionals inside the DataProvider to specifie what data must be fed into which method under test (getName().equalsIgnoreCase("methodname")).

An example:

@Test(dataProvider="countProvider")

public static void exampleSum(

@Optional("1") int startnumber, @Optional("1") int addition) {

System.out.println("adding "+startnumber+" + "+addition+"

= "+(startnumber+addition));

}

@Test(dataProvider="countProvider")

public static void exampleMultiply(@Optional("1") int todubble) {

System.out.println("multiplaying "+todubble+" gives us: "+(2\*todubble));

}

@DataProvider(name="countProvider")

public Object[][] getDataFromDataprovider(**java.lang.reflect.Method m**){

if**(m.getName().equalsIgnoreCase("exampleSum"))**{

System.out.println("\nTesting the SUM-method");

return new Object[][] {

{ 1, 1 },

{ 2, 5 },

{ 1000, 234 }

};}

else{

System.out.println("\nTesting the SUM-method");

return new Object[][] {

{ 1 },

{ 2 },

{ 1000 }

};

}

### (2C) Parameters Using The DataProvider (ITest)

This allows grouping of tests (when setting up environments or additional harnesses this can become helpful). There are here two ways of doing it: one with the use of an XML-file and the other without. First the information will be given about the one using the XML-file:

In the @Test annotation you add the groups that those tests belong to. This is done using the “groups=”groupname””. This can than later be read using the ITestContest in the DataProvider-method and then .getInclidedGroups() to set up a loop. When this is done, make a XML-file and run it through the XML-file.

An example:

@Test(dataProvider="countProvider", **groups = {"A"}**)

public static void exampleSum(@Optional("1") int startnumber, @Optional("1") int addition) {

System.out.println("adding "+startnumber+" + "+addition+" = "+(startnumber+addition));

}

@Test(dataProvider="countProvider", **groups = {"B"}**)

public static void exampleMultiply(@Optional("1") int todubble) {

System.out.println("multiplaying "+todubble+" gives us: "+(2\*todubble));

}

@DataProvider(name="countProvider")

public Object[][] getDataFromDataprovider(**org.testng.ITestContext thegroups**){

System.out.println("inside the dataprovider and c is: "+thegroups);

Object[][] groupArray = null;

for (**String group: thegroups.getIncludedGroups()**){

System.out.println("the group is: "+ group);

if(**group.equalsIgnoreCase("A")**){

System.out.println("\nTesting the SUM-method");

groupArray= new Object[][] {

{ 1, 1 },

{ 2, 5 },

{ 1000, 234 }

};

break;

}

else if(**group.equalsIgnoreCase("B")**){

System.out.println("\nTesting the SUM-method");

groupArray= new Object[][] {

{ 1 },

{ 2 },

{ 1000 }

};

break;

}

}

return groupArray;

}

The XML-File for testing this is:

<?xml version="1.0" encoding="UTF-8"?>

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

<suite name="test-parameter">

<test name="example1">

<groups>

<run>

<include name="A" />

</run>

</groups>

<classes>

<class

name="pakageneme.nameofclass" />

</classes>

</test>

<test name="example2">

<groups>

<run>

<include name="B" />

</run>

</groups>

<classes>

<class

name=" pakageneme.nameofclass " />

</classes>

</test>

</suite>

The other method only uses the Class-file itself and can be run like a normal test. For this you need to replace the “ITestContext” with ITestNGMethod and the “.getIncludedGroups()” with “getGroups()”.

An example:

@Test(dataProvider="SearchProvider",groups="A")

public void testMethodA(String author,String searchKey) {

{

System.out.println("Welcome ->"+author+" Your search key is->"+searchKey);

}

}

@Test(dataProvider="SearchProvider",groups="B")

public void testMethodB(String searchKey) throws InterruptedException{

{

System.out.println("Welcome ->Unknown user Your search key is->"+searchKey);

}

}

@DataProvider(name="SearchProvider")

public Object[][] getDataFromDataprovider(**ITestNGMethod c**){

Object[][] groupArray = null;

for (String group : c.**getGroups()**) {

if(group.equalsIgnoreCase("A")){

groupArray = new Object[][] {

{ "Guru99", "India" },

{ "Krishna", "UK" },

{ "Bhupesh", "USA" }

};

break;

}

else if(group.equalsIgnoreCase("B"))

{

groupArray = new Object[][] {

{ "Canada" },

{ "Russia" },

{ "Japan" }

};

}

break;

}

return groupArray;

}

}

## Handeling Cookies In Selenium Webdriver

Before we start with the Selenium part it might be good to explore what a cookie actually is (PS: not the kind you eat). A cookie is a small piece of information (usually 4kb) that is send back to the server each time the browser requests a page from the server, which is used to track your navigation through the website. This information is stored in a text-file format on your drive by the server itself. The location (path) where this information is stored is depended on the browser used.

The purpose is to generate an interaction between the user and the website so that;

-you can implement a shopping cart

-personalise a web experience

-marketing

-…

The cookie content is mainly three things:

1The name of the server the cookie is send from

2The lifespan of the cookie (Expiry)

-Session cookie=>deletes as the browser closes

-Persistent cookie=>months-years on the drive

3A value: a randomly generated unique number

Other info the cookie contains can be for example: domain, path, security status, …

There are several tools that can be used to view/edit the cookies in the browser itself. These are “Mozilla Advance cookie Manager” (for Firefox, <https://addons.mozilla.org/en-us/firefox/addon/cookie-manager/)> and “Edit this cookie” (for Chrome, <http://www.editthiscookie.com)>.

|  |  |
| --- | --- |
| Tabel1: The most used Selenium commands to handle cookies. | |
| **driver.manage().-** | **Function** |
| getCookies(); | Get all the cookies for the current domain. (Returns a list) |
| getCookieNamed(arg0); | Will retrieve a specific Cookie with a specific name |
| addCookie(arg0); | Create and add a Cookie (domain name is left black normally) |
| deleteCookie(Cookie); | Detetes a cookie, ignoring the domain name |
| deleteCookieNamed(arg0); | Deletes a cookie from the current domain/having a specific name |
| deleteAllCookies(); | Delete all cookies of current domain |

Now you are probably asking WHY this is handy for me to use in Selenium. Well using information from a cookie from a previous session can let you continue that session with the current browser/session. In other words it isn’t necessary to log in anymore. This can save some time (otherwise with each session you’ll need to log in again and again). You can thus go to the requested URL directly because the application will treat your browser section as authorised.

### (1)Storing The Cookie Info:

//create a webbrowser instance

//Visit the website and log into the application/site

//read the information of the cookie using()and store it to textfile Cookies.data

File file = new File("Cookies.data");

try{

// Delete old file if exists

file.delete();

file.createNewFile();

FileWriter fileWrite = newFileWriter(file);

BufferedWriter Bwrite = newBufferedWriter(fileWrite);

// loop for getting the cookie information

for(Cookie **ck** : **driver.manage().getCookies(**)){

Bwrite.write((ck.**getName()**+";"+ck.**getValue()**+";"

+ck.**getDomain()**+";"+ck.**getPath()**+";"+ck.**getExpiry()**+";"+ck.**isSecure()**));

Bwrite.newLine();

}

Bwrite.flush();

Bwrite.close();

fileWrite.close();

}catch(Exception ex)

{

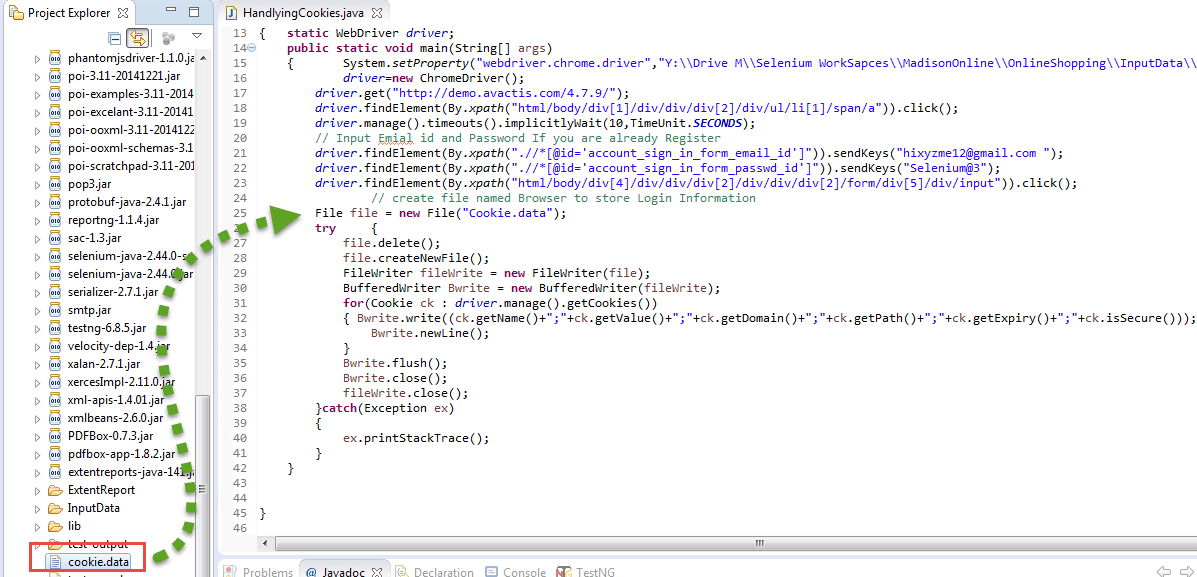
ex.printStackTrace();

}

}

}

This “Cookies.data” file will be placed into the project folder.It can be retrieved when starting a new session and lets you go directly to the wanted page without the need to log in.



### (2)Using The Stored Cookie To Log Into The Application:

//create a webbrowser instance

try {

File file = new File("Cookies.data");

FileReader fileReader = new FileReader(file);

BufferedReader Buffreader = new BufferedReader(fileReader);

String strline;

while ((strline = Buffreader.readLine()) != null) {

StringTokenizer token = new StringTokenizer(strline, ";");

while (token.hasMoreTokens()) {

String name = token.nextToken();

String value = token.nextToken();

String domain = token.nextToken();

String path = token.nextToken();

Date expiry = null;

String val;

if (!(val = token.nextToken()).equals("null")) {

expiry = new Date(val);

}

Boolean isSecure = new Boolean(token.nextToken()).booleanValue();

Cookie ck = new Cookie(name, value, domain, path, expiry, isSecure);

// This will add the stored cookie to your current session

driver.manage().addCookie(ck);

}

}

//Will tell you where something went wrong in the code

} catch(IOException e){

e.printStackTrace();

}

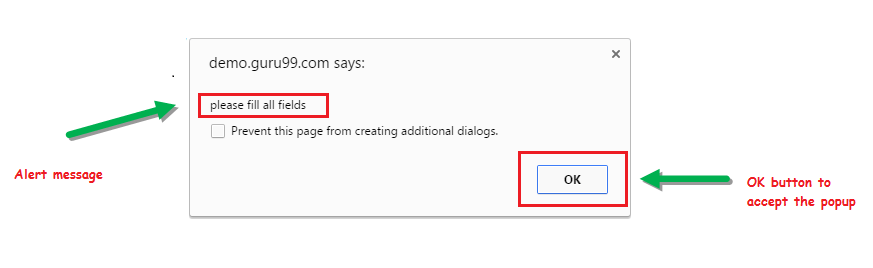
## Alert & Popup Handling

### (A)Alerts

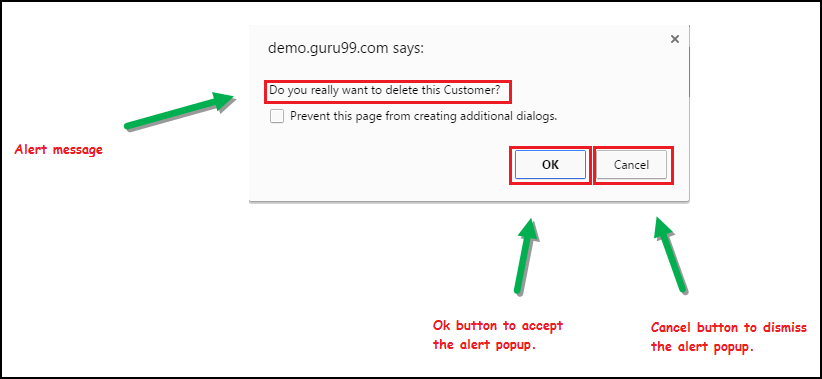
An alert is a small message box that displays an notification to give information, a warning or to ask for permission to do something.

There are 3 types:

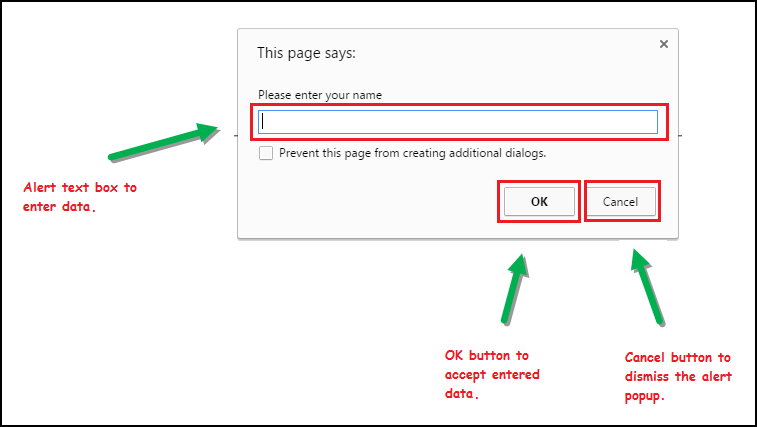
-Simple alert=>simple text+ an “OK-button” to proceed. Example from GURU99:



-Confirmation alert: simple text+ OK-button+ Cancel-button. Example from GURU99:



-Prompt Alert: simple text+ OK-button+ Cancel-button+ input-textwindow



To handle the alert you need to first switch to the alert itself. This is done using:

driver.switchTo().alert();

|  |  |
| --- | --- |
| Tabel2: The most used Selenium commands to handle alerts. | |
| **driver.switchTo().alert().** | **Function** |
| sendKeys(“texttosend”); | Sends text to the input window inside the alert |
| dismiss(); | This will click on the “Cancel”-button |
| accept(); | This will click the “OK/accept”-button |
| getText(); | This captures the alert message. |

Like in the previous automations it is wise to wait until the alert-window is properly loaded. This is primarily done using :

wait.until(ExpectedConditions.alertIsPresent());

An example of a written program (goes to site, enters wrong id and password, print text and then accepts):

driver.get("http://demo.guru99.com/V4/");

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

WebDriverWait wait= new WebDriverWait(driver,10);

driver.findElement(By.name("uid")).sendKeys("mngr30127");

driver.findElement(By.name("password")).sendKeys("EzAtAqy");

driver.findElement(By.name("btnLogin")).submit();

wait.until(ExpectedConditions.alertIsPresent());

String alertMessage=driver.switchTo().alert().getText();

System.out.println(alertMessage);

//accept alert

driver.switchTo().alert().accept();

### (B)Popup Windows

It can be that we need to switch between multiple windows to complete the whole operation. So we need a way to switch between the different windows (and ultimately move back to the main/parent window).

To get the handle of the current window you use:

driver.getWindowHandle

This will return a string-value. It can be used later on to identify the main window.

When multiple windows are in play you need a way to go over them, do the intended tasks and then close them again. So first you use a command to get the handles of all the windows. Then you create an iterator, so you can move over the different window using a loop. This looks like:

Set<String> s1=driver.getWindowHandles();

Iterator<String> i1=s1.iterator();

To switch to the needed window the command to use is:

driver.switchTo().window(“handleofwindow”);

Again here you need to have elements to wait until the elements are properly loaded. When you create a new popup-window you can use:

wait.until(ExpectedConditions.numberOfWindowsToBe(numberofwindowstoopen));

When switching to the popup- or main-window you can use the same explicit waits as when going to a new webpage.

To close a popup-window you can use:

driver.close(); //⬄ driver.quit(); for closing the session

An example of a programclass (going to a site, click to open the popup-window, fill in a text window and then close this window before closing the session):

System.setProperty("webdriver.gecko.driver", "lib/geckodriver");

WebDriver driver = new FirefoxDriver();

//setting a waits

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

WebDriverWait wait=new WebDriverWait(driver,10);

//Launching the site.

driver.get("http://demo.guru99.com/popup.php");

driver.manage().window().maximize();

//see what is the main windox

String MainWindow=driver.getWindowHandle();

System.out.println("The mainwindow is: "+MainWindow);

//clicking to open the new window

driver.findElement(By.xpath("//\*[contains(@href,'popup.php')]")).click();

wait.until(ExpectedConditions.numberOfWindowsToBe(2));

// To handle all new opened window.

Set<String> s1=driver.getWindowHandles();

Iterator<String> i1=s1.iterator();

int ofstring=s1.size();

System.out.println("number of items in windowhandles: "+ofstring);

while(i1.hasNext())

{

String ChildWindow=i1.next();

System.out.println("The window in list is: "+ChildWindow);

if(MainWindow.equalsIgnoreCase(ChildWindow)){

System.out.println("\tThis is the mainwindow!");

}

if(!MainWindow.equalsIgnoreCase(ChildWindow))

{

// Switching to Child window

driver.switchTo().window(ChildWindow);

wait.until(ExpectedConditions.visibilityOfElementLocated((By.name("emailid"))));

driver.findElement(By.name("emailid")).sendKeys("gaurav.3n@gmail.com");

// Closing the Child Window.

driver.close();

}

}

// Switching to Parent window i.e Main Window.

driver.switchTo().window(MainWindow);

driver.quit();

## OOP (Object Oriented Programming)

Do you remember in Lab one I told you it was possible to break a large program into different methods so it became reusable and more “workable”. Well the same can be done with classes. OOP is a programming method that combines data and the instructions for processing this data into objects. An object is created when a class is imitated. It has a particular state (by the variables, representing the relative aspects of an items condition) and behaviour (by the methods that give some observable effect).

An object can interact with another object (one method calling another). This characteristic is called “message passing”/”method invocation”. By doing this you can create programs with a higher complexity. It makes the code easier to handle, lowers redundancies/code repetitions,…

Object Oriented Programming has 4 main features:

### Inheritance

This feature allows you to pull (and use) code (so its methods and variables) from an existing class into another one. The class that inherits these elements is called the subclass. The class that gets inherited is the parentclass. To extend one function you use:

class NameClass extends NameParentclass{

so it will look like:

class Super {

.....

.....

}

class Sub extends Super {

.....

.....

}

Watch out: Java doesn’t support multiple inheritance (1class inheriting the features of 2 other classes). It does support single inheritance (the example above), one parent can have multiple subclasses (so parent is reusable), and multilevel (the parentclass is itself a subclass and so on). Methods that are set to private can’t be inherited.

If elements are present in both the parent as in the sub the feature of the sub will be used. This can help modify the behaviour of the subclass (also called method overriding). If you do need the feature of the parent use “super.nameofmethod/variable”.

### Polymorphism

This feature represents the fact that you can define one interface with multiple implementations. In different instances the exhibited behaviour is changed based on the data that is used. One example we have seen in inheritance (data overriding). But this can also be done within one class. The same method-name can be used, but each of these “methods” will have different variables it wants to use. This is called Method overloading. An example is:

class Overload

{

**void demo** (int a)

{

System.out.println ("a: " + a);

}

**void demo** (int a, int b)

{

System.out.println ("a and b: " + a + "," + b);

}

double demo(double a) {

System.out.println("double a: " + a);

return a\*a;

}

}

### Encapsulation/”Data Hiding”

By using the “private” control modifier in the class setup (of with global variables) these elements won’t be “visable” to other classes. This thus follow the principle of information hiding. These “private” elements can only be used by methods of the same class. Their function is executed through an “assessor method” (is made “public”).

An advantage of this is that you can split the program in different functions and when calling the class you’ll only see the “assessor methods”. So you aren’t swarmed with possible methods that you could use and are only shown those you must use. Other advantages are that when you have to change the code only the private elements can be changed, without interfering with the dataflow between classes (through the public assessors). This gives thus a smaller change of breaking the build.

### Abstraction

With this feature it is about dealing with the functionality, not the details of the operation. This is done using abstract classes and interfaces (contain only abstract methods).

A class can be made abstract by using the keyword “abstract” in the classmodifier. A class that is abstract can’t be initiated (can’t create objects with it) but must be inherited to be used. An abstractclass can contain regular methods, abstract methods or both.

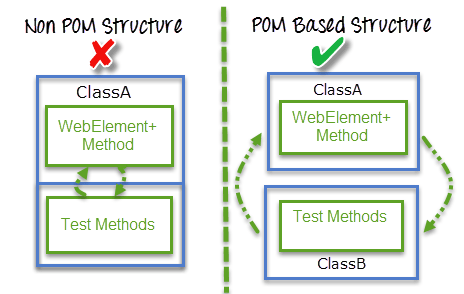
An abstract method is created like an abstract class: you use the “abstract” keyword in the method declaration. If a class contains an abstract method, it must be set to an abstract class. An abstract method contains a method signature but no body. It looks like:

abstract public void anotherMethod();

Any class inheriting the class must either override the abstact-method or become an abstract-class itself. It shows important heatores that the subclass should have and needs to override. By doing this you create an underlying line that connects the sublasses.

## Page Object Pattern (POM) By Using PageFactory

Also in WebDriver automation script it can become handy to break a bigger program into smaller components. As the test suits are adding to the pile, the code becomes very complex and harder to maintain. By creating separate classfiles for each operation we reduce the complexity/maintainability, while increasing the flexibility and readability. When doing this it becomes important to give the driver you are using to the called function (like giving a baton to the next runner in the race).



When you create a class for example the login you can call upon it from many different testfunctions. If there is a change in the login steps you only need to change 1 file. If every test had its own login imagine how bothersome it would be to go to every file and change this element over and over again. This would take up a lot of time and be highly error prone.

But what is now POM exactly? Well, it is a object Repository for web UI elements. For each “webpage” in the automation there would be a class in the file structure containing the needed steps. Keep in mind that you give each class a descriptive name that will tell you (and the poor soul that has to look at your code) an idea what it does. There are 2 was of implementing a POM (the examples are based upon the googlesearch exercise).

### (1)Regular POM

Like stated above you want to break the large project up in different logical blocks (see on what pages you have to work on or what kind of operation you need to do). These different blocks can then be made into different classes. Inside these classes you break down these into different subblocks (be made into methods). To make this executable you create one “masterfunction”. Here you place the different methods in the needed order and add if necessary extra elements. To call the different functions use:

this.nameofmethodtouse;

Of course you need to stitch together all the different classes to make one test.

Because we have to use the driver in all the methods in the method well need to initiate it using:

public NameOfTheClass(WebDriver driver){

this.driver=driver;

}

This will set the driver from the entire class.

Next it is a good idea to place all the locators of the elements in the start of the classfile. When a locator changes in the webpage we only need to change it in the start of the class (don’t need to go looking throughout the file). See back Lab1c) to see how this is done. In the start of the testmethod you place instances of the different methods to use (set as global variables). This is done using:

NameOfTheMethodToUse nameofvariable;

The method itself must be called in the testfunction by:

nameofvariable = new NameOfTheMethodToUse(driver);

Here the driver will be given to that method and allow us to use the functions of that called method with the current driver. You call the functions of the called method hereafter by:

nameofvariable.functionOfTheCalledFunction();

Small remark: it might be wise to set the methods used in the different classes as private. When making the “masterfile”/testfile. You aren’t swamed with the different methods used to create the mastermethod.

Example classfile to place text in the searchengine of Google:

public class searchandfind {

WebDriver driver;

By cleantextfield=By.id("lst-ib");

By addsearch= By.id("gs\_htif0");

By searchbutton=By.name("btnG");

**//driver initialization**

**public searchandfind(WebDriver driver){**

**this.driver=driver;**

**}**

public void clearTextfield(){

driver.findElement(cleantextfield).clear();

}

public void addSearchstring(String searchstring) {

driver.findElement(addsearch).sendKeys(searchstring);

}

public void doSearchForElement() {

driver.findElement(searchbutton).click();

}

public void placeSearchOnGoogle(String searchelement) {

//make sure textfield is clear

**this.clearTextfield();**

//fill in information

**this.addSearchstring(searchelement);**

//click on the searchbutton

**this.doSearchForElement();**

}

}

The testfile:

public class SimpleTestPOM {

WebDriver driver;

**searchandfind searchObject;**

@BeforeTest

public void open\_browser()

{

System.setProperty("webdriver.gecko.driver", "lib/geckodriver");

driver = new FirefoxDriver();

System.out.println("have started initiation");

driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);

}

@AfterTest

public void close\_browser()

{

driver.quit();

}

@Test

public void testGoogleExercise() {

driver.get("https://www.google.be/");

//create search on google site =>here place driver we are working on!

**searchObject=new searchandfind(driver);**

//can now do search

**searchObject.placeSearchOnGoogle("GUI Tests");**

}

}

### (2)Pom Using PageFactory

The main difference with the previous element is that we use here the annotation @FindBy. This annotation lets you create webelements from the start (so here we create pointers to the webelement itself and not to the locator like in the regular POM). This means you replace the:

By cleantextfield=By.id("lst-ib");

With:

@FindBy (id="lst-ib")

WebElement cleantextfield;

The @FindBy works with id, name, className, tagName, linkText and partialLinkText. Next to this we change the constructor of the driver into:

public nameoftheclass(WebDriver driver){

this.driver=driver;

**PageFactory.initElements(driver, this);**

}

The last element lets you use the created WebElements as object so you can use for example:

nameofelement.clear();

Instead of:

driver.findElement(cleantextfield).clear();

Not only will you have to type less in the end, your code becomes more logical and readable.

example of class to enter searchstring in the textbox of Google:

public class searchandfind {

WebDriver driver;

**@FindBy (id="lst-ib")**

**WebElement cleantextfield;**

**@FindBy (id="gs\_htif0")**

**WebElement addsearch;**

**@FindBy (name="btnG")**

**WebElement searchbutton;**

//driver initialization

public searchandfind(WebDriver driver){

this.driver=driver;

**//DON'T FORGET THIS!!!!!!!!!!!!!!!!!!!!!!**

**PageFactory.initElements(driver, this);**

}

public void clearTextfield(){

**cleantextfield.clear();**

}

public void addSearchstring(String searchstring) {

**addsearch.sendKeys(searchstring);**

}

public void doSearchForElement() {

**searchbutton.click();**

}

public void placeSearchOnGoogle(String searchelement) {

//make sure textfield is clear

this.clearTextfield();

//fill in information

this.addSearchstring(searchelement);

//click on the searchbutton

this.doSearchForElement();

}

}

The testfile remains the same as the precious one

## Cumcumber

## Tasklogger