TABLE OF CONTENTS

1. Setup

1.1. Have a Java project available

The first step in creating any sort of automated tests is to create an infrastructure where to gather all your test code. Meaning you need to have a code project, in which to perform all the setup you need and where to store your test code, test files, utilities code and files, and any other resource you might need in order to run the tests that you want.

In this book i will go over all sorts of examples to demonstrate the features of Selenium and how you can use this library to help with your testing. All the test code, utilities, Selenium drivers and HTML code used for demonstration are stored in a Maven project called ‘selenium-tutorial’. I will not go over what Maven is, or how to use it, but instead i will provide their website, for you to read about it in case you are new to Maven:

https://maven.apache.org/index.html. The Maven project can be downloaded from its Github location: https://github.com/iamalittletester/selenium-tutorial.

A few words about how the demo project is setup: being a Maven project, upon creation a default structure was created for it, structure which reflects in folders on the computer where you store the project. The folders and their purpose are described below:

- **src** → **main** → **java**: here you will find utility code, which is not test code but is used in the tests. In this project, in this folder, you will find several other folders, that group the utilities based on their purpose. One of these folders, 'browser', will contain the class where all the browser initialization and shutting down is stored. Another folder, 'tutorialSolution', contains a folder, 'pages' where the PageObjects classes are stored. They are used for creating handles to the HTML elements that you interact with in a test.
- **src** → **main** → **resources**: here will be all the .html files which will be open in the browser in the tests. On these .html files will the interactions demonstrated in this book occur. They will be open in the browser, just as a regular webpage is, with the only difference that these html files are not deployed anywhere on the web. They are just standalone web pages.
- **src** → **test** → **java**: this is where the tests will be stored, each in their corresponding folder. The folders help group tests by functionality or feature under test, so that you can easily find them.
- **src** → **test** → **resources**: here, in the 'browserBinaries' folder, the driver binary files, described in this chapter, are stored, which are used for interacting with the web browsers.

1.1.1. Importing the required dependencies

In order to create and run Java Selenium tests, some external libraries need to be imported into the project. Since the examples in this book are from a Maven project, in its' 'pom.xml' file, the following dependencies are declared:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>${selenium.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
```

```
    <artifactId>commons-lang3</artifactId>
    <version>${commons.version}</version>
  </dependency>
```

The first dependency, junit-jupiter-api, is used for creating and running test classes and test methods, setting up some actions that need to be performed before anything else in the class or before each method runs, or setting up some actions that need to be performed after all the class or after each method finishes its' run.

All things Selenium are accessible due to the import of the 'selenium-java' library.

The remaining import, 'commons-lang3' is used for other operations in the test project, like helping determine the OS on which tests are running. It is a library which contains a collection of helping methods, like processing of Strings, generating random Strings, easily working with Files, and so on.

1.2. Choose the OS to support

A decision you need to make is what OSs you want to be able to run your tests on. You don't need to decide at the beginning, when you are creating your test framework, but you can add to the list of supported OSs as you progress in your testing. This is because the setup for a newly supported OS is not complicated and does not take too much time to accomplish.

When you decide on what OSs to run the tests on, think about who will run these tests: you or some colleagues of yours, or maybe they will be run on a remote machine, by a Jenkins a runner.

It is important to know what these supported OSs for the tests are, because in the setup process you will need to use some files called 'drivers' (described in subchapter 1.4.) for allowing browser interactions. These are specific to each OS. What you don't need to concern yourself with is which version of the OS you will use. For example, just think about whether Windows is supported, instead of considering whether Windows 10, 8, 7, XP and so on will be needed.

In the setup chapters of this book, for the sake of example, the supported OSs will be considered: Linux, Mac, Windows.

1.3. Choose the browsers to support

Based on the supported OSs, you can also choose which browser to support on each of them. Some browsers will of course work on only one OS (Internet Explorer), whereas some have cross-OS support (like Chrome or Firefox). This again is needed for knowing which browser drivers need downloading.

For each combination of OS + browser, you will need to download a corresponding binary file. That will be discussed in the next subchapter.

For the purpose of example, in this book, the following browsers will be supported: for Windows: Chrome, Firefox; for Linux: Chrome, Firefox; for Mac: Chrome. The setup for each of these combinations of OS and browsers will be described further in this book.

1.4. Download the drivers for the supported browsers

1.4.1. What are the drivers

When you are interacting with a browser by means of a Selenium test, you are not directly giving commands to the browser. An intermediate step exists in your communication with the browser, which involves a file called a 'driver'. The driver will receive the commands you want to send to the browser, and forward these to the browser. The response from the browser based on the commands you sent will also be received by the driver and forwarded back to your tests. Simply put, the communication between the test code and browser is: test → driver → browser (when issueing a browser interaction command) and browser → driver → test (for receiving the response of the interaction command).

The driver files are specific to each browser and the operating system they are running on. In order to run Selenium tests, you will need to download the driver files corresponding to the browser/operating system you are aiming to test on.

1.4.2. Where to download the drivers from

Each browser binary is maintained by different projects, as you can see on the official Selenium documentation page, the 'Consumer browsers' and 'Specialised browsers' sections: https://seleniumhq.github.io/docs/start.html#getting_started_with_webdriver. For the purpose of this book, the Mozilla driver, called 'geckodriver', and the Chrome driver are downloaded and setup in the project. Their download locations are:

- geckodriver: https://github.com/mozilla/geckodriver/releases
- chromedriver: https://chromedriver.storage.googleapis.com/index.html

From each url you can download all the released versions of these drivers. However you need to take into account browser-driver compatibility: each driver version works with only specified browser versions. Ideally always use the latest driver version and latest browser version. In some cases, due to constant browser upgrades, you might encounter some strange errors in your tests, due to the fact that the latest browser you installed does not work with the driver you have in your project. If such a situation occurs, simply visit the driver download location and download the latest version. If you are already at the latest version, constantly go

back to the download page, as the maintainers are surely working on a new version which will be compatible with your browser.

At each of these download locations, you will find the driver files corresponding to the Windows, Linux and Mac OSs. For some versions, you will also have the possibility to choose between driver files for 32 or 64 bits operating systems.

In order to have your tests run on any machine, the driver files should be stored in the test project. This way you avoid the need for the drivers to be physically present on all the machines where the tests will run. The suggested location for storing the driver files is a dedicated folder in the 'src/test/resources' location in your project. In the test project used across this book, the driver files are stored in the following location: src/test/resources/browserBinaries. This location will need to be specified in the browser initialization methods described in the following chapter.

For the examples in this book, the following drivers were downloaded: Chrome driver for Windows, Linux and Mac, and Mozilla geckodriver for the same OSs. Now, since the driver files for Linux and Mac have the same name for Chrome, the one for Mac will be renamed to 'chromedriverMac'. Similarly, for geckodriver, the filename for the Mac driver will be changed to 'geckodriverMac'. Therefore, the setup for this books contains, in the src/test/resources/browserBinaries folder, the following files: chromedriver.exe (for Windows), chromedriver (for Linux), chromedriverMac (for Mac), geckodriver.exe (for Windows), geckodriver (for Linux) and geckodriverMac (for Mac). The Windows and Linux versions of these files are those for the 64 bit operating systems.

Chapter References

Porter, B., Zyl, J. and Lamy, O. (2019). *Maven – Welcome to Apache Maven*. [online] Maven.apache.org. Available at: https://maven.apache.org/index.html [Accessed 1 Mar. 2019].

2. Starting and closing the browser

Once the drivers needed in the test project were downloaded and stored in the project, the browser related methods can be created. This chapter focuses on creating methods for starting the browsers and closing them. It makes sense for all these methods to be in the same place. Therefore a browser dedicated class should be created, which will hold the start/close and other browser related methods that will be used across the test project. This chapter will also discuss how not to hardcode browsers in tests, how to size the browser window and strategies for how often you need to open the browser in your tests.

2.1. Hardcoding the browser versus not

Whenever you write Selenium tests, one of the first steps in each test represents opening the browser. Ideally you want to be able to run the same test on several browsers seamlessly. You don't normally want to have a test configured in such a way that it only runs on one browser. That means you need to have a way of generating a browser instance based on a parameter you will provide when the test starts to run, for those cases when you want to run the same test on different browsers.

There might be situations when you do need to only run the test on one specified browser. Such situations might include: the application you are testing only runs on that browser, as per the requirements; there is a bug in another browser or in the test framework for a specific browser, and you cannot run certain test steps except on another, specified browser; your requirement clearly specifies what browser to test on and it is the only one required. For this scenario, you will hardcode the browser in the test.

If you have no requirement which restricts you to running tests on just one browser, than you should use the approach of not hardcoding the browser in the test. The good news is that, if, let's say you want to run a test on 3 different browsers, you don't need to write 3 tests, one for each browser. Instead you will write the test not taking into account what the browser is. You are only interested in the test steps, and what actually needs to be done in a test. The test will be 'browser unaware'.

In this book, an approach is used where a dedicated class is created for storing all the methods which start up the browser. In this class, called 'BrowserGetter', you will find a method for starting up the Chrome browser (named 'getChromeDriver()'). If your test requires you to only use Chrome in the test class, you will call this 'getChromeDriver()' method, which will start your Chrome browser. In the same class, 'BrowserGetter', you will also find a method for starting up the Firefox browser (named 'getFirefoxDriver()'), which you will call from a test where Firefox is the mandatory browser to be used. These are explained in subchapter 2.2.

For those tests where you won't hardcode the browser, a method called 'getDriver()' can also be found in the 'BrowserGetter' class. Based on a System property that you can change before running any test, it allows you to create an instance of a browser corresponding to the value of the System property.

2.2. Creating the methods for starting the specific browsers

Whether you want to hardcode the browser in the tests or not, you will need a method for starting each browser. When you don't hardcode the browser in a test, based on a parameter, the method corresponding to the parameter value will be called to start up the specified browser. Check subchapter 2.4. for details on non hardcoded browsers.

For now, in this subchapter, the methods for starting the Chrome and the Firefox browsers are described. The class which will gather all browser start/closing method is, in this book, called 'BrowserGetter', and its' location in the test project is: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/browser/BrowserGetter.java.

2.2.1. Chrome

As i mentioned in the previous chapter, you need to consider on what OSs the tests will be run. Let's say you decided to use Chrome on Windows, Linux and Mac. A new method for starting Chrome will be created in the 'BrowserGetter' class which will start the browser on all these three selected OSs. The entire method is found here, and the explanation on what it does follows. The name of the method is 'getChromeDriver()' and it returns a value of type WebDriver.

```
public WebDriver getChromeDriver() {
        if (!IS_OS_WINDOWS && !IS_OS_LINUX && !IS_OS_MAC) {
            throw new RuntimeException("Could not initialize browser due to unknown operating
system!");
        }
        if (IS_OS_WINDOWS) {
            setProperty("webdriver.chrome.driver",
"src/test/resources/browserBinaries/chromedriver.exe");
        }
        if (IS_OS_LINUX) {
            setProperty("webdriver.chrome.driver",
"src/test/resources/browserBinaries/chromedriver");
        }
        if (IS_OS_MAC) {
            setProperty("webdriver.chrome.driver",
"src/test/resources/browserBinaries/chromedriverMac");
        }

        WebDriver driver = new ChromeDriver();
        driver.manage().window().maximize();
        return driver;
    }
```

What this suggested method does:

- It uses some fields defined in the SystemUtils class from the Apache Commons Library, to determine what the OS is. These fields are of boolean type, and they will return true, as follows: IS_OS_WINDOWS will return true when the OS on which the code where you are using this field is Windows; IS_OS_LINUX will

return true if the OS on which the code is executing is Linux; IS_OS_MAC will be true when the code is executing on Mac. In order to use the SystemUtils class, the following import must be declared in the import section of the 'BrowserGetter' class:

```
import static org.apache.commons.lang3.SystemUtils.*;
```

- The browser initializing method first checks whether the operating system is anything but the desired ones (Windows, Linux, Mac), and if it is, an exception is thrown and no browser will be initialized. This is because in the setup of this project only drivers corresponding to these operating systems are downloaded and stored, and so no other driver can be used to start the browser, except for those for Windows, Linux and Mac. There is no point in doing anything else if the OS detected is not the expected one, that is why an exception will be thrown.

```
if (!IS_OS_WINDOWS && !IS_OS_LINUX && !IS_OS_MAC) {
        throw new RuntimeException("Could not initialize browser due to unknown operating system!");
    }
```

- If the operating system is one of the expected ones, a system property called 'webdriver.chrome.driver' needs to be set and to point to the location of the driver file. If the operating system is Windows, the system property needs to be set to the location of the .exe driver file; if the OS is Linux, the system property needs to point to the location of the Linux driver file; same for the Mac OS. Setting the system property is done using the method 'setProperty' from Java's System class, for which the following import must be declared in the import section of the 'BrowserGetter' class:

```
import static org.apache.commons.lang3.SystemUtils.*;
```

- So, for Windows, setting the 'webdriver.chrome.driver' system property to the location of the Chrome Windows driver file is done:

```
if (IS_OS_WINDOWS) {
        setProperty("webdriver.chrome.driver", "src/test/resources/browserBinaries/chromedriver.exe");
    }
```

- For Linux, setting the 'webdriver.chrome.driver' system property to the location of the Chrome Linux driver file is done:

```
 if (IS_OS_LINUX) {
          setProperty("webdriver.chrome.driver",
"src/test/resources/browserBinaries/chromedriver");
      }
```

- For Mac, the corresponding code is:

```
if (IS_OS_MAC) {
          setProperty("webdriver.chrome.driver",
"src/test/resources/browserBinaries/chromedriverMac");
      }
```

- After successfully setting the 'webdriver.chrome.driver' system property, the Chrome driver will be initialized, by the following code:

```
WebDriver driver = new ChromeDriver();
```

- This will open the browser and store the driver instance to a WebDriver variable. As you noticed, on the right side of this assignment a new ChromeDriver() instance is created, but on the left side, the type of variable to which this instance is stored is of WebDriver. The WebDriver is the interface which is implemented (indirectly) by all the specific drivers, like ChromeDriver, FirefoxDriver, etc. All of the tests should use a driver variable of type WebDriver. You will see in the method for instantiating a browser based on a parameter why this is useful.
- The size of the browser however will not be the same every time it opens. It depends on how the browser works, and usually, on the size of the last opened window. Therefore, in order to keep the tests consistent, you should maximize the browser window, once the browser is open. Otherwise, you can't know for sure how large the size of the window will be when a test is running, and how much of the page you want to test is in view, or how the page renders (if its' design is responsive). Maximizing the window once there is a window open is done by the following code:

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

When the Chrome initialization method needs to be called from a test, the 'driver' variable (declared to store the new ChromeDriver() instance) will be needed in the test. Selenium methods need a valid driver instance in order to interact with the browser. Therefore, the method created for starting Chrome needs to return a valid driver instance, which, in the test, will be stored to a WebDriver variable or field.

The last line of code from the initialization method will return the newly initialized ChromeDriver instance:

```
return driver;
```

In a test, in the declaration section, the WebDriver field needs to be created:

```
private WebDriver driver;
```

Calling the ‘getChromeDriver()’ method from a test can be done like this:

```
driver = browserGetter.getChromeDriver();
```

In subchapter 2.6. there are some hints on where in the test class to place the browser initialization method call.

Keep in mind that the Chrome initialization method presented in this subchapter is a suggestion, since it takes into account the OSs on which the tests will be run. If, let’s say, you know for a fact that you will only run the tests on Windows, you can use a simplified version of this method:

```
public WebDriver getChromeDriver() {
        setProperty("webdriver.chrome.driver",
"src/test/resources/browserBinaries/chromedriver.exe");
        WebDriver driver = new ChromeDriver();
        driver.manage().window().maximize();
        return driver;
    }
```

This simplified version of the initialization method will: set the ‘webdriver.chrome.driver’ system property to point to the location of the ‘chromedriver.exe’ file (so the Windows driver file), initialize the ChromeDriver() and store the resulting driver instance to a WebDriver variable, maximize the window, then return the driver instance, to be used in tests that call this method.

2.2.2. Firefox

Starting the Firefox browser is done in a similar fashion to the Chrome one. In my GitHub project, in the same class where the Chrome related method is declared, namely BrowserGetter, you can find the ‘getFirefoxDriver()’. It looks like:

```
public WebDriver getFirefoxDriver() {
       if (!IS_OS_WINDOWS && !IS_OS_LINUX && !IS_OS_MAC) {
          throw new RuntimeException("Could not initialize browser due to unknown operating
system!");
       }
       if (IS_OS_WINDOWS) {
          setProperty("webdriver.gecko.driver",
"src/test/resources/browserBinaries/geckodriver.exe");
       }
       if (IS_OS_LINUX) {
          setProperty("webdriver.gecko.driver",
"src/test/resources/browserBinaries/geckodriver");
       }
       if (IS_OS_MAC) {
          setProperty("webdriver.gecko.driver",
"src/test/resources/browserBinaries/geckodriverMac");
       }

       WebDriver driver = new FirefoxDriver();
       driver.manage().window().maximize();
       return driver;
   }
```

In this method, the first check that is made is related to the OS on which the test will run. If it is an unkown OS (so not Windows, Linux or Mac) a new Exception will be raised and no browser will be started, since the code project only stores driver files for the known OSs. In this case, there are 3 driver files, corresponding to Firefox and to the OSs: Windows, Linux, Mac. These driver files, named ‘geckodriver.exe’ (for Windows), ‘geckodriver’ (for Linux) and ‘geckodriverMac’ (for Mac) can be found in the project, in the location ‘src/test/resources/browserBinaries’.

For each of the supported OSs, the path to the corresponding geckodriver location needs to be set as the value of the ‘webdriver.gecko.driver’ system property. For Windows, this looks like:

```
if (IS_OS_WINDOWS) {
          setProperty("webdriver.gecko.driver",
"src/test/resources/browserBinaries/geckodriver.exe");
       }
```

The corresponding code for Linux is:

```
 if (IS_OS_LINUX) {
```

```
        setProperty("webdriver.gecko.driver",
"src/test/resources/browserBinaries/geckodriver");
    }
```

And of course, similarly for Mac:

```
 if (IS_OS_MAC) {
        setProperty("webdriver.gecko.driver",
"src/test/resources/browserBinaries/geckodriverMac");
    }
```

Once the 'webdriver.gecko.driver' system property is pointing to the correct location of the required driver for each OS, a new driver instance can be spawned. This time the driver is of type 'FirefoxDriver':

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

At this point the Firefox browser will be started. The size of the browser window is not going to be always the same, each time it opens. It depends on the size of the windows as it was last opened/closed. Therefore, to ensure good results and the same size each time the tests run, after the browser window is open, you should maximize it with the following code:

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

At this point, the setup of the browser is done, and you can return the 'driver' instance to the calling test:

```
return driver;
```

In a test class, in the declaration section, the WebDriver field needs to be created:

```
private WebDriver driver;
```

Calling the 'getFirefoxDriver()' method from a test can be done like this:

```
driver = browserGetter.getFirefoxDriver();
```

If you are not interested in enabling the browser for all OSs, you could use a simplified version of the ‘getFirefoxDriver()’ method. Let’s say you only want to run the tests on Windows. In this case, the checks regarding the OS on which the tests run can be removed, and the remaining code of the method can be:

```
public WebDriver getFirefoxDriver() {
      setProperty("webdriver.gecko.driver",
"src/test/resources/browserBinaries/geckodriver.exe");
       WebDriver driver = new FirefoxDriver();
       driver.manage().window().maximize();
       return driver;
   }
```

2.3. Creating the method for starting a browser based on a parameter

Not hardcoding the browser in your tests is the best option to choose from. It gives you flexibility in terms of running the same test, with a single small change, across all your supported browsers. The small change i am refering to involves a System property, whose value will say which browser the test is currently running on. The idea is that, right before a test is about to run, the tester sets a value to this property, corresponding to what browser they want the test to run on.

This mechanism needs to be implemented in your tests project. Otherwise said: you will need to create a separate browser instantiation method, similar to those for Chrome and Firefox, but with some additional logic. It needs to identify the value of the System property which says what browser to run the tests on, and based on that value, it will start up the corresponding browser. So there are two points we need to address here: how will the System property be set, and what does the browser initialization method look like?

Let’s start with the first one, namely setting the System property. Let’s say, for the sake of example that the name of this property is ‘browser’. This is the name that will be used in the examples in this book. There are several ways of setting this System property. I will start with the one i do not recommend, and i will also tell you why. You can create a property file (which is nothing more than a text file, with ‘key’ = ‘value’ pairs of text, and with a .properties extension). Then, in the browser initialization method, the value corresponding to the ‘key’ property is read from this file, which will give the name of the browser needed in the test. Such a key/value pair from this file would look something like:

```
browser=Chrome
```

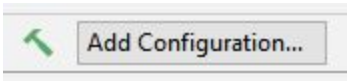
This means that whenever somebody needs to change the browser on which the test runs, they will just change the value in this file, before starting the test run. It sounds rather simple. However there is a major disadvantage with this approach: the person running this tests is not the only person who needs to work on this project. There are surely at least two testers who need to work with the same code, and who need to start the browser, for running some tests (maybe not the same tests as the first tester). And since testers will also use a Version Control System, the same file will be commited with different values for the browser property, whenever one of the people running the tests will change its' value.

This might not be a significant disadvantage, unless you need to make sure you run certain tests on certain browsers, which will lead to you constantly checking the values from the property file, to make sure they contain the expected values. The huge disadvantage however can be observed if the same tests need to be run on a CI. If you have several CI jobs where the same tests need to be run, but on a different browser, you cannot do this, since the CI jobs will pull the content from the Version Control repository for each job (ideally, since you always want to run the latest version of your test code). Once the property file is commited with a value for the browser, all the CI jobs started after that, which pull code from the repository, will use that particular value. Let's say somebody commited the property file with the value of Chrome for the browser, at 2pm. Jobs started at 2:01pm, 2:01pm, and so on, before any new commit is made, will take the value of the browser commited at 2pm. Therefore, it is very difficult to manage using several browsers by different CI jobs with this approach, as you would require to either: not have your jobs pull code from the repository once the property file you pulled contained the expected browser, or to commit the file with the expected value for a certain CI job right before that CI job is started (each time a job needs to run).

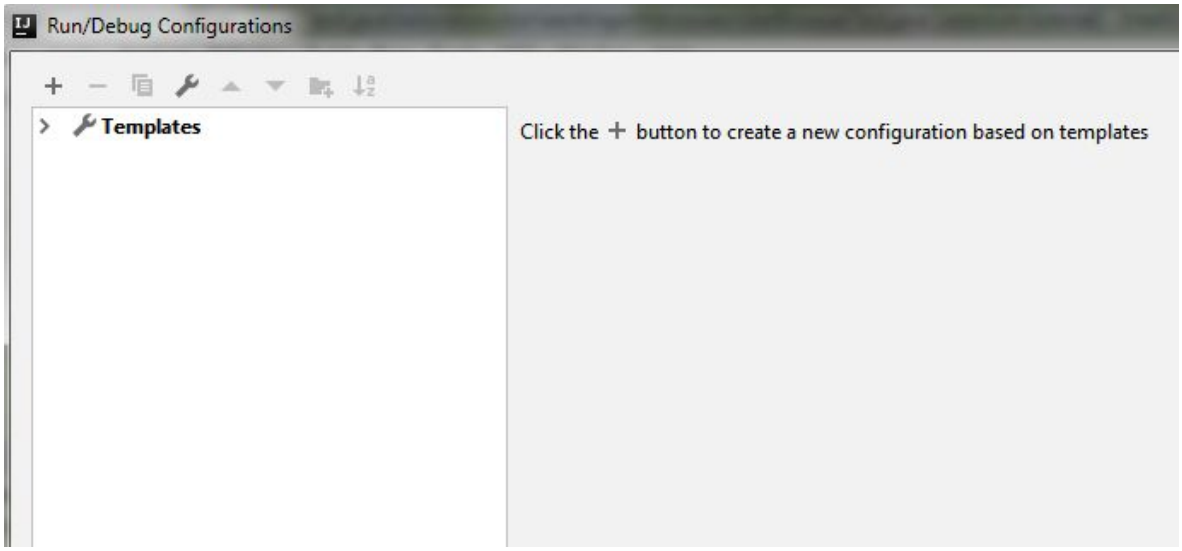
The better alternative is to provide the browser name as a System property (called 'browser' in this book), which can be provided via the IDE or via the command line, depending on how the test will be run.

2.3.1. Setting the property from the IDE

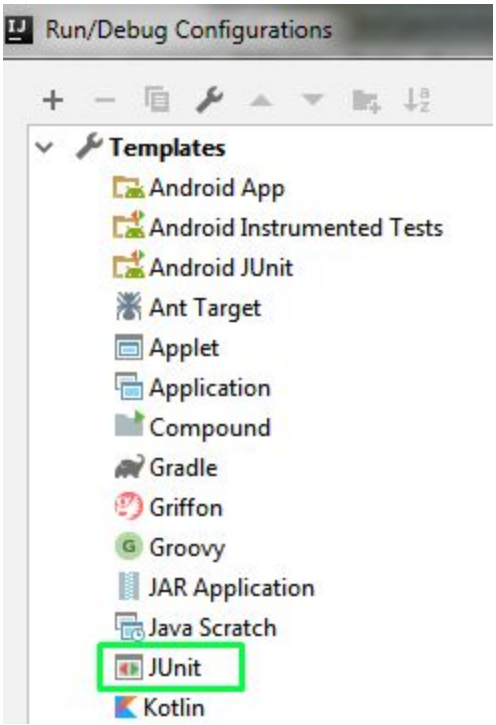
When you are running the tests directly from the IDE (not from the command line), the 'browser' System property can easily be set from the 'Run/Debug Configurations' screen (don't forget i am referring to IntelliJ in this book). On the top right side of the IDE screen, you will find the Configurations section, which, if it hasn't been accessed before, displays the 'Add Configuration…' label.



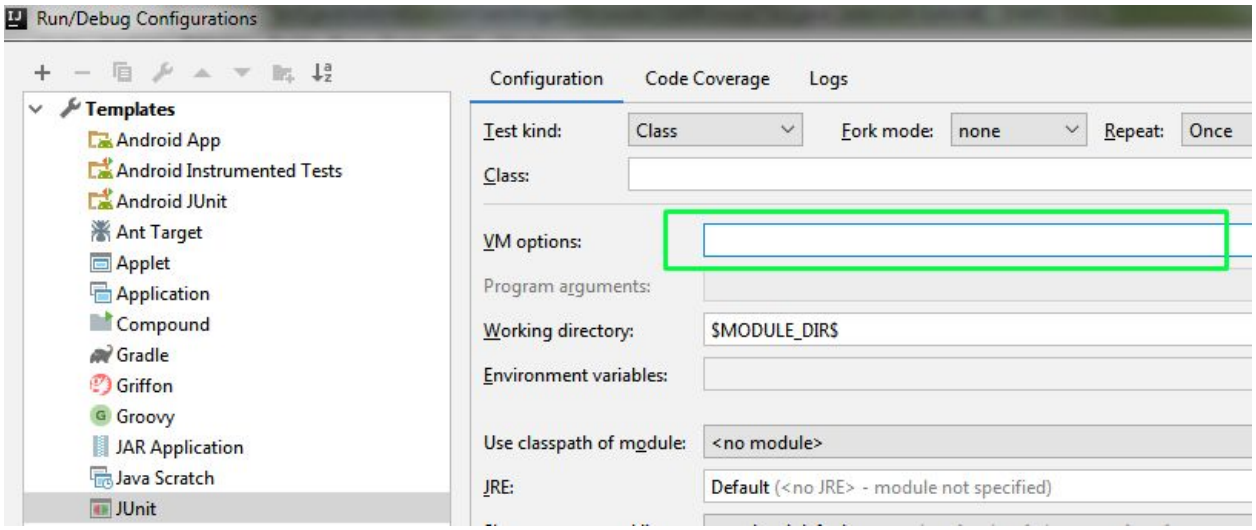
Clicking on the label will open the editing screen. From there you can change certain aspects of running the tests, by setting certain values of certain attributes from the Templates created in the IDE. First you need to click on the 'Templates' label.



This will open the left hand menu, where you can see different technologies to change the settings for, including JUnit. Since in this book i am using only JUnit, this is the template that needs to changed in order to set the ‘browser’ property, by clicking on the ‘JUnit’ label.



This will open the edit section on the right hand side of the already opened screen. The only field you are interested in at this point is the ‘VM options’ one.



In this field, you only need to type ‘-Dbrowser=nameOfDesiredBrowser’. ‘-D’ signals that you want to set a property, whose name is typed right after ‘-D’. In this example, the name of the property you are setting is ‘browser’. The value of the property is the text after the ‘=’ character. So, for example, if you want to run your tests on Chrome, you can type ‘-Dbrowser=Chrome’ into the ‘VM options’ field.

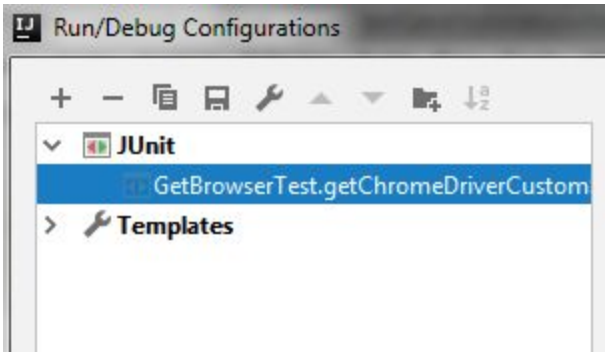
As you can see in the how the browser initialization method code, if you want to start the Chrome browser, you don’t need to type the name of the browser with specific case. Whether a letter is in upper case or lower case, that does not matter, since the browser initialization method knows how to deal with the case. It is however important that the part before the ‘=’ is written exactly as in the screenshot above.

After setting the value of the ‘browser’ property, click the ‘Apply’, then the ‘OK’ buttons. Every test that will run from the moment you set this property will have access to this property if it needs it. That means, that if you have some tests where the browser is hardcoded, you are not interested in this property, but it is available to the tests. All those tests that will use the browser initialization method based on a System property will also have access to it, and will use it to create the browser instance specified by this property.

If you want to change the browser so that any future test runs on a different browser, you can again open the Configurations screen. If you already ran some tests, you will notice that on the left side, in the menu, above the ‘Templates’ menu entry you have another menu entry called ‘JUnit’. If you expand this menu item, you will some other entries as part of this one. Their name corresponds to the test classes or test methods you already ran. Now, if you only want to change the browser for one or several (but not all) of these entries (test methods or test class),

you can just: click on the corresponding entries, edit the value of the browser name, and click the Apply and OK buttons. This way, only the newly changed methods or classes will use the new browser you specified, while the remaining ones, and any future ones, will use the value set at the beginning.

What happens if you want every test that runs from now on to use a different browser, even those which are already displayed in the menu, under the 'JUnit' menu entry? In order to change the browser, you first need to delete all these entries from the 'JUnit' menu entry (the one above the 'Templates' menu entry), by clicking the '-' icon for each entry.



Once that is done, you need to go back to the Template entry corresponding to JUnit, and just change the name of the browser to the one you need, click Apply and click OK, then just run your tests.

2.3.2. Setting the property from the command line

If you are running your tests via Maven and the command line, either on your local machine or on a remote CI, you will have to type a specific command, along the lines of:

```
mvn test ...
```

Of course you can specify all kinds of other aspects in the command, like profiles to run, specific tests to run, and so on. Similarly you can specify what the browser should be, for those tests which will need to start a browser based on the 'browser' property. Similarly to what you needed to type in the IDE VM options field, in the command line you will need to add the following text to your command: '-Dbrowser=browserName'. Therefore, the command for running tests will become, for Chrome:

```
mvn test -Dbrowser=Chrome ...
```

Or, for Firefox:

```
mvn test -Dbrowser=Firefox ...
```

2.3.3. The browser initialization method

The method used for starting the browser based on the value of the ‘browser’ System property that you now set, can be found here, and is explained right after the code:

```
public WebDriver getDriver() {
      switch (System.getProperty("browser").toLowerCase()) {
         case "chrome" :
            System.out.println("Chrome was chosen!");
            return getChromeDriver();
         case "firefox" :
            System.out.println("Firefox was chosen!");
            return getFirefoxDriver();
         default:
                  throw new RuntimeException("Unsupported browser! Will not start any
browser!");
      }
   }
```

This method returns a WebDriver instance. This can be, in this example, either a Chrome or Firefox instance, which is possible since WebDriver is the interface implemented by both ChromeDriver() (used for starting Chrome) and FirefoxDriver() (used for starting Firefox).

Retrieving the value of the system property named ‘browser’ is done on the first line of the method, by calling ‘System.getProperty()’. If the value is not specified (either via the IDE or the command line, depending on how the tests are configured), a NullPointerException will be thrown. That is a hint that you forgot to set the ‘browser’ property, or that you misspelled the name of the property.

If the value of the property was read, a comparison will be made, between the lowerCase value of the property, and a String value representing the browsers the tests support. So, the property value is read. It can be, for example: Chrome, chrome, CHROME, ChROME. It does not matter what type of case is used for each letter. The value of property will be lowerCased and compared to the string ‘chrome’ (all lowercase) on the second line of code in the method. This allows for some flexibility when writing the browser name property, as you don’t nee to adhere strictly to a case for each letter. You just need to make sure the property values are from the supported set of values (in the examples in this book these supported values are chrome and firefox).

As you notice, there is a ‘switch/case’ construct in the code. Based on the property value from the ‘switch’, each ‘case’ treats what browser to be open. If the property, in lower case,

equals ‘chrome’, inside the corresponding ‘case’ block, first there is a message displayed in the console, just for informational purposes, that the Chrome browser will be initialized. Then, the ‘getChromeDriver()’ method is called, and the result of calling this method will be returned by the current method (‘getDriver()’). This means, that if the System property in lower case was ‘chrome’, the ‘return getChromeDriver();’ is the last piece of code executed from this method, and the method exits, returning the Chrome driver instance.

Similarly, in the second ‘case’ block from the above code, if the lower case value of the ‘browser’ property is ‘firefox’, a message is displayed in the console that the Firefox browser will be initialized. Then, the ‘return getFirefoxDriver();’ code will call the method for initializing the Firefox driver, and make the browser initialization method successfully exit by returning a WebDriver instance (represented by the newly open Firefox driver instance). If the ‘browser’ property value in lower case was ‘firefox’ this will be the last line of code that will execute from this method.

If the ‘browser’ property was specified, but its value in lower case was none of the already mentioned ones (‘chrome’ or ‘firefox’), it means that either there was a misspelling, or that whoever tries to run the test wants to do so on an unsupported browser. For this situation instead of initializing one of the supported browsers, it’s best to not initialize any. This is just so that the person who is trying to run the tests is aware that they did not specify the correct browser name. In the code above, to accomplish this, a Java RuntimeException is thrown, with a message reflecting the situation (that the browser specified is unsupported).

To sum it up, the above method can provide one of the following four outcomes: a NullPointerException if the System property with name ‘browser’ has not been set, a ChromeDriver instance, a FirefoxDriver instance, or a RuntimeException if the value of the ‘browser’ property is not ‘chrome’ or ‘firefox’.

Calling this method from a test class requires to first create a variable or field of type WebDriver, to which the result of calling the method will be stored. In this case, a field is created:

```
private WebDriver driver;
```

The actual call to the method is done:

```
driver = browserGetter.getDriver();
```

This way, the browser is started, and the WebDriver instance stored in the ‘driver’ variable gives allows browser interactions from your tests.

2.4. Custom window sizes for starting the browser

It is possible that not all of your tests require your browser to be in maximized size, but instead in a custom one. If you need the ability to set your browser window to a particular size, according to your requirements, you could proceed as follows: create a method similar to that corresponding to the browser you are interested in (let's say Chrome, hence the method is 'getChromeDriver()'), but instead of having the 'driver.manage().window().maximize();' line of code inside the new method, you could replace it with code which sets the window size to the desired one. This code is:

```
driver.manage().window().setSize(new Dimension(width, height));
```

The 'setSize()' method will set the window to a size that you specify as a parameter to this method call, based on a width and a height you need. This method requires a parameter of type Dimension, therefore passing the size to the 'setSize()' method is done by creating a new Object of type Dimension, by calling the constructor of the Dimension which takes two int parameters: the width and height. In order to use the 'Dimension' constructor in the BrowserGetter class, the following import needs to be made, in the class' import section:

```
import org.openqa.selenium.Dimension;
```

This step where the window size is set is the only step different from the 'getChrome()' method created in subchapter 2.2.1. The method for initializing a custom size browser window is:

```
public WebDriver getChromeDriverCustomSize(int width, int height) {
      if (!IS_OS_WINDOWS && !IS_OS_LINUX && !IS_OS_MAC) {
         throw new RuntimeException("Could not initialize browser due to unknown operating
system!");
      }
      if (IS_OS_WINDOWS) {
         setProperty("webdriver.chrome.driver",
"src/test/resources/browserBinaries/chromedriver.exe");
      }
      if (IS_OS_LINUX) {
         setProperty("webdriver.chrome.driver",
"src/test/resources/browserBinaries/chromedriver");
      }
      if (IS_OS_MAC) {
         setProperty("webdriver.chrome.driver",
```

```
"src/test/resources/browserBinaries/chromedriverMac");
        }

        WebDriver driver = new ChromeDriver();
        driver.manage().window().setSize(new Dimension(width, height));
        return driver;
    }
```

When this method is called from a test, the size of the window you want to open must be specified. This browser initialization method requires two parameters to be passed to it when it is called: the first one is an int parameter which represents the desired width of the window, whereas the second one is an int which represents the desired height. These two parameters will be send to the 'setSize()' method which resizes the browser window, based on these parameters. An example of a method call, where a browser window of width 460 and height 640 is needed is:

```
driver = browserGetter.getChromeDriverCustomSize(460, 640);
```

2.5. Closing the browser

Every time you are finished with running a Selenium test, which obviously opens at least a browser window, you also need to close the browser. Otherwise you would be stuck with hundreds of open browsers/driver instances, which will lead to poor performance of the system you are running your tests on, not to mention the mess caused by these hundreds of open browser windows.

Therefore, just as you opened the browser/driver instance, you will need to perform some cleanup, which implies closing any open browser window which was initialized through the driver. This means, from Selenium you can close those browser instances which were started by running one of the following code snippets:

```
driver = browserGetter.getChromeDriver();
```

```
driver = browserGetter.getFirefoxDriver();
```

```
driver = browserGetter.getDriver();
```

Make sure when you start up the browser, you are storing the result of calling a start up method to a variable of type WebDriver. Otherwise, if you fail to do so, there is no way of controlling the browser instance from Selenium.

Once you are finished with the test step execution, you will need to close all the browser windows that the test opened. Luckily there is a method in Selenium for closing all open windows at once, namely 'quit()'. The usage of this method is as follows:

```
driver.quit();
```

Keep in mind that the 'quit()' method is called on a specific WebDriver instance. Therefore it will only close those browser instances/windows associated with the same WebDriver instance. So, if, let's say two sets of tests are running in parallel, each opening a new browser instance which is stored to a WebDriver field, calling the 'quit()' method on the driver instance created by the first test will not close the browser instances spawned by the second one. That is because, even if you run the same test twice at the same time, in parallel, a new WebDriver field is created for each test run. Closing all browsers from the first running test will only close the browsers started by that test.

There is also a different method, 'closed()', which will only close the current browser window. This means it should not be used, unless you only want to close a single browser instance. Its' usage is:

```
driver.close();
```

Sometimes you might run into the following exception in your test:

```
org.openqa.selenium.NoSuchSessionException: Session ID is null. Using WebDriver after calling quit()?
```

This means that either the browser crashed while running a test, or that somewhere the 'quit()' or 'close()' methods were called, which caused the browser to close while some test steps still needed to run. If the browser crashed unexpectedly and you are on Windows, make sure to close any remaining 'chromedriver.exe' or 'geckodriver.exe' processes that might still be running. It the browser crashes frequently while running tests, and these processes are not closed, there will be a huge number of them (running) which will make the browser run very slow at one point.

2.6. How often will you open the browser

You have now implemented methods for opening the browser, depending on your needs. The next question to ask yourself is: how to call these methods. Of course this will be done from a test class, but where in the test class will the call to these methods be found?

This all depends on what your testing goal is. Normally, for each test method that will be run, a 'clean' browser session should be worked with. What that means is really up to the tests you are running. When the 'new ChromeDriver()' or 'new FirefoxDriver()' code is executed (from within the browser initialization methods), a new browser instance is created, which has no cookies stored, and has no history. It is a session just like those you normally get after clearing the cache and cookies of the browser you are working with.

Let's see some situations and how to handle opening the browsers in their case:

1. Each test method needs a clean browser session, which means: no cookies, no cache.
   Solution 1: Open the browser in a @BeforeEach method. This annotation from JUnit signals the fact that the method annotated with it will be run each time a test method will be run, before the test method runs. So, before a test method runs, a @BeforeEach method will run, which opens the browser. But because the browser opens before each test method, it also need to be closed, right after the test method finished running. Therefore, in a method annotated with an @AfterEach annotation, the browser will be closed. The @AfterEach method will run every time a test method runs, right after it is finished executing, hence no browser instance is left open once the test ran.
   Pros: clean browser session for each test.
   Cons: opening the browser takes some time. Not minutes, but a good amount of seconds. That means that for each test method that requires this approach, this additional browser spawning time is added to the test execution time. If you have, let's say 1000 tests that need to use this approach, and let's say the browser takes 5 seconds to open, this means you have 5000 seconds of total execution time that you are wasting, which is about 83 minutes. What i mean by 'wasting' is that you are not actually executing any test steps at that time, so it is not productive time.
   Solution 2: If clearing the cookies is sufficient to ensure what your tests expect to be a 'clean state of the browser', you could try an approach where you will open the browser only once per test class, and clear the browser cookies before each test method runs. This means that opening the browser will be done in a method annotated with @BeforeAll, which is executed before any test method is executed, just once. Then, in a method annotated with @BeforeEach, the cookies will be deleted by calling 'driver.manage().deleteAllCookies();' (which is explained in subchapter 7.1.). Closing the browser will also be done, just as opening it, just once per test class, in a method annotated with @AfterAll, which runs after all the test methods were executed.
   Pros: Less openning of the browser means a lot of test execution time saved.

2. Before any test runs a setup must be executed only once which includes opening the browser (for example logging in and performing some actions).
   Solution: In a method annotated with @BeforeAll, the browser is opened, and all the needed setup is performed. In a method annotated with @AfterAll the browser is closed.

3. In a test class there is a mix of tests that require the browser and that do not require the browser at all (these tests are not front-end facing and do not interact with any webpage).
   Solution: In this case, a split should be done: all tests that require a working browser instance need to be extracted to a test class whose purpose is interacting with the needed webpages; all other tests should be moved to a test class whose purpose will be deduced by the purpose of these tests. Once this is done, the browser tests will follow one of the approaches described in paragraphs 1 or 2, depending on the requirements.

GitHub location of example code

- BrowserGetter class, with the methods for opening a browser: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/browser/BrowserGetter.java
- Browser driver files: https://github.com/iamalittletester/selenium-tutorial/tree/master/src/test/resources/browserBinaries
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialsolution/workingwithbrowsers/GetBrowserTest.java

Chapter References

Commons.apache.org. (2019). *SystemUtils (Apache Commons Lang 3.9-SNAPSHOT API)*. [online] Available at: https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/SystemUtils.html [Accessed 20 Mar. 2019].

Docs.oracle.com. (2019). *System (Java SE 10 & JDK 10)*. [online] Available at: https://docs.oracle.com/javase/10/docs/api/java/lang/System.html [Accessed 20 Mar. 2019].

Seleniumhq.github.io. (2019). *WebDriver*. [online] Available at: https://seleniumhq.github.io/docs/wd.html#browser_launching_and_manipulation [Accessed 20 Mar. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/chrome/ChromeDriver.html [Accessed 20 Mar. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at:

https://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/firefox/FirefoxDriver.html [Accessed 2 Apr. 2019].

Porter, B. (2019). *Maven – Introduction to the Build Lifecycle*. [online] Maven.apache.org. Available at: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html [Accessed 3 Apr. 2019].

Junit.org. (2019). *BeforeEach (JUnit 5.0.2 API)*. [online] Available at: https://junit.org/junit5/docs/5.0.2/api/org/junit/jupiter/api/BeforeEach.html [Accessed 9 Apr. 2019].

Junit.org. (2019). *AfterEach (JUnit 5.0.1 API)*. [online] Available at: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/AfterEach.html [Accessed 9 Apr. 2019].

Junit.org. (2019). *BeforeAll (JUnit 5.0.0 API)*. [online] Available at: https://junit.org/junit5/docs/5.0.0/api/org/junit/jupiter/api/BeforeAll.html [Accessed 9 Apr. 2019].

Junit.org. (2019). *AfterAll (JUnit 5.0.2 API)*. [online] Available at: https://junit.org/junit5/docs/5.0.2/api/org/junit/jupiter/api/AfterAll.html [Accessed 9 Apr. 2019].

3. Opening a page

The browser is now open. Your tests will now require for a page to be loaded in the open browser in order to interact with it. Whether you need to click on a button or read some page values, you need an open page to interact with. In subchapter 3.1. i will go over how to open a page (URL). Subchapter 3.2. will deal with checking that the expected page (URL) opened. And subchapter 3.3. will show how to check the page's title.

Example setup

In order to exemplify the methods for opening a page, checking that the correct URL was loaded, and checking the page title, an existing URL will be demonstrated, whose sole purpose is to be used as example. This URL is www.example.com and can be accessed on http and https, with or without the preceding 'www.'. Therefore, in this chapter, the following URLs are used, which are all valid, real ones:
http://www.example.com
https://www.example.com
http://example.com
https://example.com

The import section of the test clas (https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/openPage/OpenPageTest.java) contains the following entries:

```
import browser.BrowserGetter;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;
import org.openqa.selenium.WebDriver;

import java.net.MalformedURLException;
import java.net.URL;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.TestInstance.Lifecycle.PER_CLASS;
```

The ‘junit’ related imports are used for the test annotations and running the tests. The MalformedURLException and the URL imports are used by one of the method used for opening a URL (driver.navigate().to() with URL as parameter). The BrowserGetter class is imported in order to enable starting the browser in the @BeforeAll method, while the WebDriver import allows for the driver instance generated by the BrowserGetter to be stored to a WebDriver variable:

```
@BeforeAll
    public void beforeAll() {
        //initialize the Chrome browser here
        driver = browserGetter.getChromeDriver();
    }
```

In the @AfterAll method, the browser instance will be closed:

```
    @AfterAll
    public void afterAll() {
        driver.quit();
    }
```

3.1. Open page with ‘get()’ and ‘navigate().to()’

Opening a page once the browser has been initialized is very easy. It can be done in three ways, but the most common one is to call the ‘get()’ method and pass it the URL of the page you want to open as a String. It needs a valid ‘driver’ instance in order to be called. The usage of this method is:

```
driver.get(url);
```

The URL passed to this method needs to have the protocol specified (for example http or https). Therefore a valid URL looks like "http://theUrl" or "https://theUrl", where 'theUrl' is the valid URL of the page you want to open. If the protocol is not specified, the following Exception will be thrown and the test where the 'get()' method is used will fail:

```
org.openqa.selenium.WebDriverException: unknown error: unhandled inspector error:
{"code":-32000,"message":"Cannot navigate to invalid URL"}
```

Therefore, when you encounter such an Exception, check that the URL you specified in the 'get()' method has the correct protocol and is a valid URL.

A second way of opening a URL is to use the 'driver.navigate().to()' construct, to which you need to pass the URL you want to open as a String parameter. In this case you will also need to specify the protocol in the URL, in order for it to be valid. The usage of this approach is:

```
driver.navigate().to(url);
```

A third way of opening a URL in the browser is to use the 'driver.navigate().to()' construct, but by pasing a parameter of type URL. Creating the parameter can be done by just specifying 'new URL(theURL)', where 'theURL' is a String corresponding to a valid URL (also containing the protocol). You are basically calling the constructor from Java's URL class which takes a String as parameter in order to create a URL object. The usage is:

```
driver.navigate().to(new URL(url));
```

The above code is the easiest way to generate a new URL parameter (by only specifing the URL as a String). Consult the official documentation to see what other ways of creating a URL type parameter there are, by calling different constructors from the URL class: https://docs.oracle.com/javase/10/docs/api/index.html?java/net/URL.html

If the protocol is not specified in the url, in this last approach you will get a different type of exception, namely:

```
java.net.MalformedURLException: no protocol:
```

Note: In many examples from this book, the HTML pages used for demonstration are not deployed anywhere on the internet. Instead they are open from a location inside the test project itself. All these HTML files can be found under the following location: src/main/resources. Opening such a page in the browser will be done as follows:

```
driver.get(new File("src/main/resources/nameOfHTMLFile.html").getAbsolutePath());
```

Example

Scenario 1. Open the following page by using ‘get()’: www.example.com.

In order for the URL to be valid, the http protocol will be specified in the URL, since the requirement did not specify that we need https.

```
driver.get("http://www.example.com");
```

Scenario 2. Open the following page by using ‘get()’ on https: www.example.com

```
driver.get("https://www.example.com");
```

Scenario 3. Open the following page by using ‘get()’: example.com

In order for the URL to be valid, the http protocol will be specified in the URL, since the requirement did not specify that we need https.

```
driver.get("http://example.com");
```

Scenario 4. Open the following page by using ‘get()’ on https: example.com

```
driver.get("https://example.com");
```

Scenario 5. Open the following page by using ‘driver.navigate().to()’ with a String parameter: https://www.example.com

```
driver.navigate().to("https://www.example.com");
```

Scenario 6. Open the following page by using ‘driver.navigate().to()’ with a URL parameter: http://example.com

```
driver.navigate().to(new URL("http://example.com"));
```

3.2. Check the current URL with ‘getCurrentUrl()’

When you need to check that the current URL loaded in the browser is the expected one, you will use Seleniums’ ‘getCurrentUrl()’ method. The result of calling this method will be a String value, which you can then compare to the String representing the expected URL. The method usage is:

```
driver.getCurrentUrl()
```

Note that this method is usually used when you are somehow landing on a new page and you want to check that it is the correct one. Getting to the page is either done by clicking on a button or a link from a different page, or opening a page (with ‘get()’ maybe) which triggers some page redirects, until a specific page is reached. If such a redirect is made, when you are calling the ‘getCurrentUrl()’ method, sometimes you might be calling it too early, before the page you are interested in is actually loaded in the browser. For such cases (when there are some redirects in place), it is a good idea to combine the ‘getCurrentUrl()’ with a wait mechanism, which would change the approach from ‘checking that the current url is correct’ to ‘waiting for the correct url’. See chapter 13. for using waits.

For the basic scenarios where no redirects are made, it is fine to use the ‘getCurrentUrl()’ method to check that the current URL is correct. See the examples below.

It is also important to understand that checking that the URL is correct does not guarantee that the contents of the page have loaded properly. Aditional checks should be done to ensure that the customer facing page elements are working as expected.

Example

Scenario 1. Open the following page by using ‘get()’: www.example.com. Check that the ‘http://www.example.com/’ URL has loaded in the browser.

In order for the URL to be valid, the http protocol will be specified in the URL, since the requirement did not specify that we need https. The additional ‘/’ at the end of the expected url is present because the ‘getCurrentUrl()’ method will always return this character as the last one in the URL read from the browser.

```
driver.get("http://www.example.com");
assertEquals("http://www.example.com/", driver.getCurrentUrl());
```

Scenario 2. Open the following page by using 'driver.navigate().to()' with a String parameter: https://www.example.com. Check that the URL loaded in the browser is 'https://www.example.com/'.

```
driver.navigate().to("https://www.example.com");
assertEquals("https://www.example.com/", driver.getCurrentUrl());
```

3.3. Check the page title with 'getTitle()'

When a page is opened in a browser, you can see, in the tab corresponding to that page, a short description of what the page is all about. The description comes from an HTML present on the page, namely the <title> tag. The content of this tag is what you see in the tab heading. The <title> tag also gives the title under which you can see a page you visited, in the browser history. Checking the title of a page is not an activity you might need to do often, but for those cases when you do, you can use Seleniums' 'getTitle()' method (which returns the title as a String, whose leading and trailing whitespaces were removed):

```
driver.getTitle()
```

Example

Scenario 1. Open the following page by using 'get()': www.example.com. Check that the tile of the page is 'Example Domain'.

```
driver.get("http://www.example.com");
assertEquals("Example Domain", driver.getTitle());
```

Chapter References

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/WebDriver.html [Accessed 21 Mar. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/WebDriver.Naviga

tion.html [Accessed 21 Mar. 2019].

Docs.oracle.com. (2019). *Java SE 10 & JDK 10*. [online] Available at: https://docs.oracle.com/javase/10/docs/api/index.html?java/net/URL.html [Accessed 21 Mar. 2019].

W3.org. (2019). *HTML 5.2: 4.2. Document metadata*. [online] Available at: https://www.w3.org/TR/html5/document-metadata.html#the-title-element [Accessed 22 Mar. 2019].

GitHub location of example code

- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/openPage/OpenPageTest.java
- Test methods: getPageProperties

4, 5. Creating the WebElements

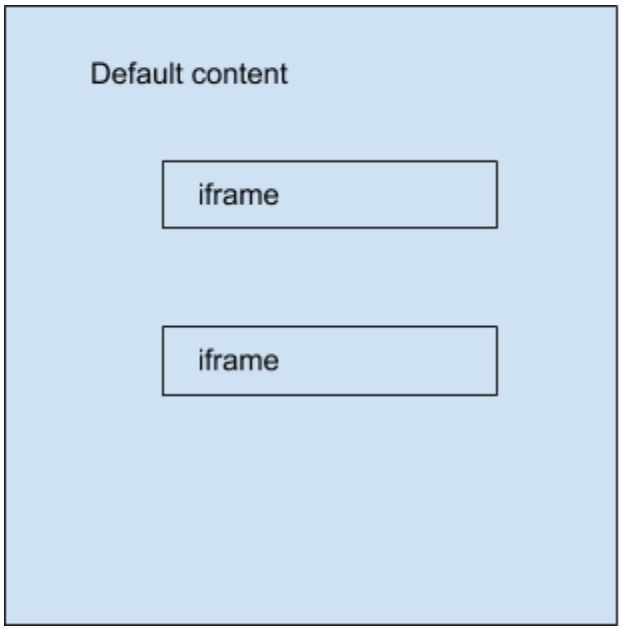
See attached powerpoint for strategies on creating WebElements.

5.3. Iframes, switchTo() and default content

So, now that you are an expert in writing CSS selectors to identify your WebElements, you want to write some new tests. You are inspecting the page you will test, identifying what WebElements you will need, and start writing the selectors. Once you have them, and the test contains all the necessary interactions with those WebElements, you run the test, confident it will pass. But instead, surprise. You get a NoSuchElementException. You double, triple, quadruple check the page, and by the looks of it, the selector is correctly written. And that is true. However, when you inspect the page further, you notice that your element is actually contained within an <iframe> tag (<iframe>...</iframe>).

In this case, the selector you wrote can be left as is. In the test however you will need to perform two additional steps: switch to the iframe that contains your WebElement before interacting with it, and switching back to the parent frame (also called default content) once you want to interact with items outside of the iframe. For both these steps you will use the 'switchTo()' method from the Selenium library.

Here is a very simple drawing representation of a document with iframes:



5.3.1. Switching to an iframe

In order to switch to an iframe, first you need to uniquely identify it. For that, in Selenium, you have three options: identify by id (or name), by index, or by a custom CSS selector.

Let's take a look at them one by one.

Iframe by id/name

As mentioned, for this approach, when the iframe tag has an ‘id’ attribute or a ‘name’ attribute, you can use them to uniquely identify the iframe for switching to it. The usage of this approach is as follows, where the id or the name that represent your <iframe> will be expressed as String parameters:

```
driver.switchTo().frame(String id);
driver.switchTo().frame(String name);
```

Below are a few HTML examples of what the <iframe> tag might look like in your application. Of course, the values for the ‘id’ and ‘name’ are made up for the purpose of demonstration.

As an example the HTML code for an iframe that can be identified by an id looks as follows:

```
<iframe id="frameWithId" ...></iframe>
```

In this case, the value of the id which needs to be passed to the ‘switchTo()’ method is the value of the ‘id’ attribute from this HTML element. And that value is ‘frameWithId’. In the test, switching to this iframe will be done as follows:

```
driver.switchTo().frame("frameWithId");
```

On the other hand, an example of the HTML code for an iframe that can be identified by a name looks as follows:

```
<iframe name="frameWithName" ...></iframe>
```

In this case, the value needed to be specified for identifying this iframe will be the value of the name attribute, namely ‘frameWithName’. In the test, switching to this iframe will be done as follows:

```
driver.switchTo().frame("frameWithId");
```

Iframe by index

If your iframe has no id or name, or if they have a different value each time the page is rendered, there are still other ways to identify it in order to access it from your tests. One of them is to identify the frame by its index.

For this approach, you need to first inspect the page and see how many <iframe> tags you have. Imagine you would store all the <iframe> tags in a Java List. The index refers to the index in the List of the <iframe> tag you are interested in, considering that indexes for Java Lists start with 0. So, let’s say you have 5 <iframe> tags on your page and you are interested in the third one. That means the index you will use for identifying your frame in the tests will be 2. Note that this approach only works if the <iframe> you are interested in has the same index each time the page opens.

In order to switch to a frame identified by index in Selenium, you will use the following syntax, where the index is represented as an int parameter:

```
driver.switchTo().frame(int index);
```

For this approach, the <iframe> tag can have whatever attributes it is allowed to have, since those do not matter in the process of identifying it.

Iframe as WebElement

This approach is used to identify the <iframe> based on a selector, just as you would for identifying a regular WebElement. For example, you could use the value of the ‘src’ attribute, or the value of the ‘class’ attribute if it exists and uniquely identifies the frame.

The syntax for switching to an <iframe> identified as a WebElement is as follows, where the parameter of type WebElement represents the selector of the frame:

```
driver.switchTo().frame(WebElement element);
```

Let’s say that the HTML code for the <iframe> you are interested in is the following one:

```
<iframe src="forFrameAsWebElement.html" ...></iframe>
```

In this case, the selector used for identifying the WebElement will be based on the ‘src’ attribute and it can be one where you interested in the ‘src’ attribute to contain the String ‘ FrameAsWebElement’. In my PageObject approach, in the class where i store the WebElements, i will create an entry for the this frame:

```
@FindBy(css = "[src*='FrameAsWebElement']") public WebElement frameAsWebElement;
```

Switching to this frame from a test would look like this, considering that ‘page’ is the PageObject class:

```
driver.switchTo().frame(page.frameAsWebElement);
```

5.3.2. Switch to defaultContent

Once you are done interacting with the iframe, you should switch back to the default content. That is the top level of the document, the one you normally interact with (unless you need to work inside an iframe). You need to switch to the default content every time you are done working in an iframe and want to continue interacting with the page that embeds the iframe, or when you need to switch to a different iframe (which is not embedded in the first iframe). In this latter case, the usage is: switch to frame 1, interact with it, switch back to default content, switch to frame 2, interact with it, switch back to default content.

By default when you start a test by opening a new page, the default content is the one you will interact with. In this case you don't need to switch to the default content.

Note that when you are working inside an iframe you do not have access to the page elements that are in the default content, just as when you are on the default content level you do not have access to the iframe elements.

Switching to the default content is done as follows:

```
driver.switchTo().defaultContent();
```

5.3. 3. Chapter examples

Example 1

Given the following HTML code:

```
<iframe id="frameWithId" src="forFrameWithId.html" style="width:100%;height:100px;" frameborder="no"></iframe>
```

And the content of the ‘forFrameWithId.html’ page which is embedded in the initial page as an iframe:

```
<!DOCTYPE html>
<link rel="stylesheet" href="https://www.w3schools.com/w3css/4/w3.css">
<body>
<div class="w3-display-middle w3-container w3-border-top w3-border-bottom w3-border-teal">
    <button id="btnForFrameWithId"
            class="w3-btn w3-border w3-teal w3-margin w3-padding-large w3-left w3-opacity">Inside
        frame
        with id
    </button>
</div>
</body>
```

Create a test that will check that the text on the button identified by the id 'btnForFrameWithId' is 'Inside frame with id'.

Solution

In the PageObject class, in this example 'BasicPage', a WebElement variable 'buttonForFrameWithId' will be created for identifying the button whose id is 'btnForFrameWithId':

```
@FindBy(css = "#btnForFrameWithId") public WebElement buttonForFrameWithId;
```

In the test class, in the field declaration section, the WebDriver instance and the PageObject class instance will be declared. These are 'driver' and 'page' namely.

```
private BasicPage page;
private WebDriver driver;
```

In the @BeforeClass method, the driver will be initialized and the page will also be initialized through PageFactory:

```
driver = browserGetter.getChromeDriver();
page = PageFactory.initElements(driver, BasicPage.class);
```

The page that will be tested will also be open, but that part is omitted.

In the test method, first a switch to the iframe will be done. The iframe containing the desired button has an id of value 'frameWithId'. Then, the button's text will be read with the 'getText()' method from Selenium, and it will be compared, through an assertion of type 'assertEquals()' to the expected text. The last line of code for this example will make the switch back to the default content. Here is the relevant code:

```
driver.switchTo().frame("frameWithId");
assertEquals("Inside frame with id", page.butonForFrameWithId.getText());
driver.switchTo().defaultContent();
```

Example 2

Given the following HTML code:

```
<!DOCTYPE html>
<body>
<iframe id="frameWithId" src="forFrameWithId.html" style="width:100%;height:100px;"
frameborder="no"></iframe>
<iframe src="forFrameWithIndex.html" style="width:100%;height:100px;"
```

```
frameborder="no"></iframe>
<iframe src="forFrameAsWebElement.html" style="width:100%;height:100px;"
      frameborder="no"></iframe>
</body>
```

And the content of the frame whose value for the 'src' attribute is 'forFrameWithIndex.html' and is embedded in the initial page as an iframe:

```
<!DOCTYPE html>
<link rel="stylesheet" href="https://www.w3schools.com/w3css/4/w3.css">
<body>
<div class="w3-display-middle w3-container w3-border-top w3-border-bottom
w3-border-cyan">
   <button id="btnForFrameWithIndex"
                  class="w3-btn w3-border w3-cyan w3-margin w3-padding-large w3-left
w3-opacity">Inside
      frame with index</button>
</div></body>
```

Create a test that will check that the text on the button identified by the id 'btnForFrameWithIndex' is 'Inside frame with index'.

Solution

In the PageObject class, in this example 'BasicPage', a WebElement variable 'buttonForFrameWithIndex' will be created for identifying the button whose id is 'btnForFrameWithIndex':

```
@FindBy(css = "#btnForFrameWithIndex") public WebElement buttonForFrameWithIndex;
```

In the test class, in the field declaration section, the WebDriver instance and the PageObject class instance will be declared. These are 'driver' and 'page' namely.

```
private BasicPage page;
private WebDriver driver;
```

In the @BeforeClass method, the driver will be initialized and the page will also be initialized through PageFactory:

```
driver = browserGetter.getChromeDriver();
page = PageFactory.initElements(driver, BasicPage.class);
```

The page that will be tested will also be open, but that part is omitted.

In the test method, first a switch to the iframe will be done. The iframe containing the desired button does not have a name or an id. Therefore, we can identify it based on its' index. Since this is the second iframe on the page, and since we are talking in Java, it will have an index of 1.

Then, the button's text will be read with the 'getText()' method from Selenium, and it will be compared, through an assertion of type 'assertEquals()' to the expected text. The last line of code for this example will make the switch back to the default content. Here is the relevant code:

```
driver.switchTo().frame(1);
assertEquals("Inside frame with index", page.butonForFrameWithIndex.getText());
driver.switchTo().defaultContent();
```

Example 3

Given the following HTML code:

```
<!DOCTYPE html>
<body>
<iframe id="frameWithId" src="forFrameWithId.html" style="width:100%;height:100px;"
frameborder="no"></iframe>
<iframe src="forFrameWithIndex.html" style="width:100%;height:100px;"
frameborder="no"></iframe>
<iframe src="forFrameAsWebElement.html" style="width:100%;height:100px;"
    frameborder="no"></iframe>
</body>
```

And the content of the frame whose value for the 'src' attribute is 'forFrameAsWebElement.html' and is embedded in the initial page as an iframe:

```
<!DOCTYPE html>
<link rel="stylesheet" href="https://www.w3schools.com/w3css/4/w3.css">
<body>
<div class="w3-display-middle w3-container w3-border-top w3-border-bottom w3-border-deep-purple">
   <button id="btnForFrameAsWebElement"
                class="w3-btn w3-border w3-deep-purple w3-margin w3-padding-large w3-left w3-opacity">Inside frame as  WebElement
   </button>
</div>
</body>
```

Create a test that will check that the text on the button identified by the id 'btnForFrameAsWebElement' is 'Inside frame as WebElement'.

Solution

In the PageObject class, in this example 'BasicPage', a WebElement variable 'buttonForFrameWithName' will be created for identifying the button whose id is 'btnForFrameAsWebElement':

```
@FindBy(css = "#btnForFrameAsWebElement") public WebElement
buttonForFrameWithName;
```

In this particular case, because the iframe we need does not have a name or id, and we don't want to identify it by index, we can write a CSS selector for identifying it based on its 'src' attribute. In the 'BasicPage' class, we will create another WebElement variable dedicated to the <iframe> tag. Its' selector refers to the 'src' attribute containing the String 'FrameAsWebElement'.

```
@FindBy(css = "[src*='FrameAsWebElement']") public WebElement frameAsWebElement;
```

In the test class, in the field declaration section, the WebDriver instance and the PageObject class instance will be declared. These are 'driver' and 'page' namely.

```
private BasicPage page;
private WebDriver driver;
```

In the @BeforeClass method, the driver will be initialized and the page will also be initialized through PageFactory:

```
driver = browserGetter.getChromeDriver();
page = PageFactory.initElements(driver, BasicPage.class);
```

The page that will be tested will also be open, but that part is omitted.

In the test method, first a switch to the iframe will be done. Then, the button's text will be read with the 'getText()' method from Selenium, and it will be compared, through an assertion of type 'assertEquals()' to the expected text. The last line of code for this example will make the switch back to the default content. Here is the relevant code:

```
driver.switchTo().frame(page.frameAsWebElement);
assertEquals("Inside frame as WebElement", page.buttonForFrameWithName.getText());
driver.switchTo().defaultContent();
```

GitHub location of example code

- PageObject class, section 'iframes page':
  https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/BasicPage.java
- Test class:
  https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/iframesSolution/IframesTest.java
- Test methods: frameById, frameByIndex, frameAsWebElement
- HTML code:
  https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/withIframes.html
  https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/forFrameWithId.html
  https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/forFrameWithIndex.html
  https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/forFrameAsWebElement.html

Chapter References

W3.org. (2019). *HTML 5.2: 4.7. Embedded content*. [online] Available at: https://www.w3.org/TR/html/semantics-embedded-content.html#elementdef-iframe [Accessed 15 Feb. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/ [Accessed 26 Feb. 2019].

6. Interacting with the WebElements

Now that you know how to get a handle to the elements of an HTML page, you are ready to interact with them. Interaction means, depending on the type of element, either clicking on it, typing in it, or reading some of its' properties. Below i will describe what these interactions are, what type of elements they are suited or supported for, and we will go over some examples for each of them.

Example setup

There will be code examples for each type of WebElement interaction, examples which you can find in the GitHub project. The setup for these tests are described in this sub chapter.

All of the tests from this chapter can be found in the ElementInteractionTest class (https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/elementInteractionSolution/ElementInteractionTest.java). For each interaction method (click, sendKeys, clear, getText, getAttribute, getCssValue) there is a corresponding test method with the same name, that demonstrates different scenarios where these methods can be used. There is also another test method (select) which demonstrates how to work with dropdowns.

In the import section, several Selenium classes are imported, as follows:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.Color;
import org.openqa.selenium.support.PageFactory;
import org.openqa.selenium.support.ui.Select;
```

The WebDriver class is used for being able to start and close the browser, and interact with it. The PageFactory class is imported for initializing the PageObject class that will be used in these examples. The WebElement class is imported because it holds the interaction methods (except for the dropdown methods, which can be found in the Select class that is the last

import). The Color class is imported as it will be used in retrieving the color properties of HTML elements by means of the ‘getCssValues()’ method.

At class level, in the test class, you will need to create the fields: ‘browserGetter’ for opening and closing the browser, ‘page’ corresponding to the PageObject class where the WebElements used by this test are stored, driver for interacting with the browser.

```
private final BrowserGetter browserGetter = new BrowserGetter();
private WebElementInteractionPage page;
private WebDriver driver;
```

Before any test can be run, the browser needs to be initialized (started), by calling the ‘getChromeDriver()’ method from within the @BeforeAll method. In this same method, once the driver was initialized, the page will also be initialized, by calling the ‘PageFactory.initElement()’ method. Once this method runs, the browser will be up and running, and the WebElements stored in the PageObject class can be used.

```
@BeforeAll
   public void beforeAll() {
      //initialize the Chrome browser here
      driver = browserGetter.getChromeDriver();
      //initialize page object class
      page = PageFactory.initElements(driver, WebElementInteractionPage.class);
   }
```

The browser will only be closed after the entire test class was run. This is done in the @AfterAll method:

```
@AfterAll
   public void afterAll() {
      driver.quit();
   }
```

Before each test method is run, the page the tests will run on is open, in the @BeforeEach method. This is to ensure a ‘clean’ state of the page for each of the tests run (since the same elements might be interacted with by different tests, which might change the state of these elements). The page used for this example is ‘interactions.html’ (https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/interactions.html).

```
@BeforeEach
   public void beforeEach() {
```

```
    driver.get(new File("src/main/resources/interactions.html").getAbsolutePath());
  }
```

The PageObject class where you can find all the WebElements used by chapter 6 examples is: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/WebElementInteractionPage.java.

6.1. click()

Possibly the most common type of interaction you will perform in a Selenium test is the 'click' one, as there are plenty of HTML elements to click on, some of which are extremely common on webpages: links (<a> tags), buttons (<button> tags), <input> tags which can represent a larger set of elements (including checkboxes or radio buttons), audio or video content (<audio> and <video> tags), and so on. Clicking will only work if the elements you are interacting with do not have a 'hidden' attribute.

The purpose of clicking on an HTML element is to trigger an event, like: the checkbox being selected, the radio button being selected, the page to be refreshed, a new page to be open, a popup to appear, a new element to be displayed, a form to be submitted to the backend and so on. Many times, clicking an item will also need a corresponding assertion, to check that the clicking triggered the desired event. For example, when checking a checkbox, by using the 'getAttribute()' method described in section 6.5 you can check that the state of the checkbox is 'checked'. Or, by clicking on a link, by using the 'getCurrentUrl()' method, you can check that the correct URL was opened.

The usage of Seleniums' 'click()' method used for clicking on a page element, is as follows:

```
clickableElement.click();
```

Here, clickableElement is the WebElement variable representing the handle to the desired HTML element to be clicked.

Example

Scenario 1. Checking a checkbox which was not checked. Then, unchecking it.

The HTML code corresponding to the checkbox is:

```
<input type="checkbox" name="checkboxToClick">A checkbox<br>
```

The corresponding WebElement is:

```
@FindBy(css = "[name='checkboxToClick']") public WebElement checkboxToClick;
```

Clicking on the checkbox, action which will check it, from Selenium, is done as follows:

```
page.checkboxToClick.click();
```

Clicking on the same checkbox for a second time will uncheck it:

```
page.checkboxToClick.click();
```

Scenario 2. Checking a radio button. Once it is checked, it cannot be unchecked.
The HTML element which represents the radio button is:

```
<input type="radio" id="radioButtonToClick">The radio button
```

The WebElement corresponding to the radio button is:

```
@FindBy(css = "#radioButtonToClick") public WebElement radioButtonToClick;
```

Clicking on the radio button, action which will check it, from Selenium, is done as follows:

```
page.radioButtonToClick.click();
```

Scenario 3. Clicking on a button which will trigger a new text label to be displayed on the page.
The HTML code corresponding to the button in this example is:

```
<button id="buttonToClick" ng-click="clickButton()" class="w3-btn w3-padding
w3-teal">Button</button>
```

The corresponding WebElement to be used in the test is:

```
@FindBy(css = "#buttonToClick") public WebElement buttonToClick;
```

Clicking the button by using Selenium's 'click()' method is done as follows:

```
page.buttonToClick.click();
```

Scenario 4. Clicking on a link which opens a new page in the same window/tab.

The HTML code which represent the link (an <a> tag) is:

```
<a id="linkToClick" href="https://imalittletester.com/">Link</a>
```

The corresponding WebElement to be used in the test is:

```
@FindBy(css = "#linkToClick") public WebElement linkToClick;
```

The test step which clicks on this link is:

```
page.linkToClick.click();
```

GitHub location of example code

- PageObject class, section 'elements for clicking': https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/WebElementInteractionPage.java
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/elementInteractionSolution/ElementInteractionTest.java
- Test methods: click
- HTML code: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/interactions.html

Chapter References

W3.org. (2019). *HTML 5.2: 4.10. Forms*. [online] Available at: https://www.w3.org/TR/html5/sec-forms.html#elementdef-input [Accessed 15 Feb. 2019].

W3.org. (2019). *HTML 5.2: 4.10. Forms*. [online] Available at: https://www.w3.org/TR/html5/sec-forms.html#elementdef-button [Accessed 15 Feb. 2019].

W3.org. (2019). *HTML 5.2: 3. Semantics, structure, and APIs of HTML documents*. [online] Available at: https://www.w3.org/TR/html5/dom.html#elements-in-the-dom [Accessed 15 Feb. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/ [Accessed 27 Feb. 2019].

6.2. sendKeys()

Another kind of interaction you will use is typing in an HTML element. To get an idea what kind of HTML elements support being typed into, refer to the first table on the following page: https://www.w3.org/TR/html5/sec-forms.html#elementdef-input. This describes the various types of <input> tags that allow typing. This behavior applies only to some <input> tags, which have certain values for their 'type' attribute.

In this table, if you look at the 'Control type' and see, for a specified 'Keyword' (first column) that the control type is textfield, you will know that an <input> tag with type equal to the value of that keyword allows for typing. For example, for the keyword 'tel' you can see that the control type is a text field. Therefore, if, in your test you encounter an <input> tag with a 'tel' 'type' attribute, you will be able to type in it. This HTML element will look like:

```
<input type="tel">
```

Another type of HTML element you can type into is a <textarea>, which is basically just a multiline text field, but which can be customized in many ways, including setting a minimum or maximum required number of characters to type, or providing a sort of hint regarding what to type into it.

Whichever of these HTML elements you are typing into, the Selenium method used for this action is 'sendKeys()' and its' usage is (where 'textToType' is a String parameter):

```
nameOfElement.sendKeys("textToType");
```

In case you need to check that the text was typed correctly into the field, you will use the 'getAttribute()' method, described in section 6.5. You will compare the resulting String from calling the 'getAttribute()' on the WebElement you are typing into for the 'value' attribute to the expectedString. For example, let's say you typed "abcd" into a text field whose corresponding WebElement is 'textFieldElement'. The check for this text field is:

```
assertEquals("abcd", textFieldElement.getAttribute("value"));
```

Example

Scenario 1. In the given <input> element, with ‘type’ attribute ‘text’, type the String “coffee”. The HTML element corresponding to this <input> tag is:

```
<input type="text">
```

The corresponding WebElement is:

```
@FindBy(css = "[type='text']") public WebElement textInput;
```

Typing the String “coffee” into this field is done as follows:

```
page.textInput.sendKeys("coffee");
```

In order to check that the text was typed correctly in the field, you could write the following assertion:

```
assertEquals("coffee", page.textInput.getAttribute("value"));
```

Scenario 2. The following <textarea> element is given:

```
<textarea maxlength="200" placeholder="12345"></textarea>
```

This text field has a ‘placeholder’ attribute, with a value of ‘12345’. This represents a hint the user will see when looking at the text field. When the user starts typing the hint disappears and whatever the user types into the field is what will be seen in the field.

The corresponding WebElement variable is:

```
@FindBy(css = "textarea") public WebElement textarea;
```

The requirement is to make the text field show the text “1234567890”. This is accomplished by typing the String “1234567890” into the field:

```
page.textarea.sendKeys("1234567890");
```

Checking that the correct text was typed into the field is done as follows:

```
assertEquals("1234567890", page.textarea.getAttribute("value"));
```

Scenario 3. After having run scenario 2, the <textarea> input field now displays the value that was typed into it, namely “1234567890”. What happens if you call the ‘sendKeys()’ method again, with the same text, on this element?

```
page.textarea.sendKeys("1234567890");
```

The text in the field becomes:

```
assertEquals("12345678901234567890", page.textarea.getAttribute("value"));
```

Further examples on <textarea> typing and text can be found in chapter 6.4. Example Scenario 8.

GitHub location of example code

- PageObject class, section ‘elements for typing’: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/WebElementInteractionPage.java
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/elementInteractionSolution/ElementInteractionTest.java
- Test methods: sendKeys
- HTML code: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/interactions.html

Chapter References

W3.org. (2019). *HTML 5.2: 4.10. Forms*. [online] Available at: https://www.w3.org/TR/html5/sec-forms.html#elementdef-input [Accessed 28 Feb. 2019].

W3.org. (2019). *HTML 5.2: 4.10. Forms*. [online] Available at: https://www.w3.org/TR/html5/sec-forms.html#the-textarea-element [Accessed 28 Feb. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/ [Accessed 28 Feb. 2019].

6.3. clear()

In the previous section, example Scenario 3, typing the same text for the second time into a field just appended the text typed the second time to the text typed the first time. In order to clear the text field after the first typing, you can use Seleniums' 'clear()' method. This way, the text typed the second time will be the only text displayed in the field. The usage of this method is:

```
webElement.clear();
```

The 'clear()' method empties the text field. Calling this method on a field, even if it is empty, is possible, but it won't accomplish anything, since the field is already clear.

Using this method is useful in situations when you want to check that for a multitude of values typed into a field a certain error message is displayed (right after typing, without the need to submit the page). In this case, you can type the first String you are interested in, check the displayed error message, clear the field, type the second String, check the error message, clear the field, and so on.

Example

Scenario 1. In the given <input> element, with 'type' attribute 'text', type the String "coffee" once, check that the text was typed correctly, clear the text field, type the String "coffee" again and check that the text in the field is now "coffee".

The HTML element corresponding to this <input> tag is:

```
<input type="text">
```

The corresponding WebElement is:

```
@FindBy(css = "[type='text']") public WebElement textInput;
```

Typing, checking, and clearing the field to accomplish the requirement of this scenario are done as follows:

```
page.textInput.sendKeys("coffee");
assertEquals("coffee", page.textInput.getAttribute("value"));
```

```
page.textInput.clear();
page.textInput.sendKeys("coffee");
assertEquals("coffee", page.textInput.getAttribute("value"));
```

Scenario 2. The following <textarea> element is given:

```
<textarea maxlength="200" placeholder="12345"></textarea>
```

This text field has a 'placeholder' attribute, with a value of '12345'. This represents a hint the user will see when looking at the text field. When the user starts typing the hint disappears and whatever the user types into the field is what will be seen in the field.

The corresponding WebElement variable is:

```
@FindBy(css = "textarea") public WebElement textarea;
```

The requirement is to type the text "1234567890" and check that the value was typed correctly. Then, the field needs to be cleared and the same text to be typed again. Another check needs to be done, to make sure the text currently displayed in the text field is "1234567890":

```
page.textarea.sendKeys("1234567890");
assertEquals("1234567890", page.textarea.getAttribute("value"));

page.textarea.clear();
page.textarea.sendKeys("1234567890");
assertEquals("1234567890", page.textarea.getAttribute("value"));
```

GitHub location of example code

- PageObject class, section 'elements for typing': https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/WebElementInteractionPage.java
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/elementInteractionSolution/ElementInteractionTest.java
- Test methods: clear
- HTML code: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/interactions.html

Chapter References

W3.org. (2019). *HTML 5.2: 4.10. Forms*. [online] Available at: https://www.w3.org/TR/html5/sec-forms.html#elementdef-input [Accessed 28 Feb. 2019].

W3.org. (2019). *HTML 5.2: 4.10. Forms*. [online] Available at: https://www.w3.org/TR/html5/sec-forms.html#the-textarea-element [Accessed 28 Feb. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/ [Accessed 28 Feb. 2019].

6.4. getText()

When you need to check the 'text' of a page element, you will use Seleniums' 'getText()' method. But what does this 'text' really mean? When it comes to a link, you are interested in the link label, which is the text you click on when you click on the link. For a button, the text is the text you see on the button.

The 'getText()' method called on a WebElement variable will return a String which represents all the text found between the opening and closing tag corresponding to that WebElement. For example, if your WebElement represents a link (an <a> tag) and its' corresponding HTML element does not contain any interior tags, the links' text will represent all the text found between the opening and closing <a> tag (all the <a> tags' attributes are omitted):

```
<a …>This is the text that will be returned by the getText() method</a>
```

The usage of the 'getText()' method, applied to a WebElement, is as follows, and returns a String value:

```
webElement.getText();
```

Calling the 'getText()' method on this <a> tag will return the following text:

```
This is the text that will be returned by the getText() method
```

Let's say that the link is included (nested) in a tag which also has some text:

```
<div>
    This text too
    <a …>This is the text that will be returned by the getText() method</a>
</div>
```

Calling the ‘getText()’ method on the WebElement corresponding to the tag will return all the text found between the opening and closing tag, with all the whitespaces found between these tags. As you can see from the HTML example, there is a new line between the text ‘This text too’ and the <a> tag, which is also returned when calling the ‘getText()’ method.

Therefore, the ‘getText()’ method returns all the text between an opening and closing tag corresponding to an HTML element, including all the text of its’ nested children. Each of those children can have nested tags themselves.

There is one interesting HTML element for which using ‘getText()’ might not work as you expect: <textarea>. Remember from chapter 6.2. that you can type into such an element. That will make the text the element displays to be the ‘text of the element’. But in reality, the text that you type can be retrieved by using the ‘getAttribute()’ method, not the ‘getText()’ one, because the text you type is not stored between the opening and closing tag of this element. Calling the ‘getText()’ method on the <textarea> will retrieve whatever text was placed in the HTML definition of the element. In other words, ‘getText()’ will retrieve whatever text you see in the <textarea> element when you open the page, before typing anything into it. If no such text was assigned to the element (there is no text between the <textarea> and </textarea> tags, the opening and closing ones), ‘getText()’ will retrieve an empty String. If the developer implemented this tag so that there is some text between the opening and the closing tags (for example: <textarea>Some text here</textarea>), this text will be retrieved by calling the ‘getText()’ method, no matter what you type into this field. For examples, see Scenario 8 below.

Example

The following HTML code is given:

```
<div id="getTextOuterDiv">
   <h3>getText()</h3>
   <div id="getTextInnerDiv">
      <img id="getTextImage"
src="https://farm2.staticflickr.com/1854/42514697560_1740d1993f_n.jpg"
          width="100"
          height="100"
          alt="This is the module with id">
      <div>
         <h5>H5 text</h5>
         <a href="https://www.example.com">Link text</a>
```

```
        <button type="button">Button text</button>
        <select>
          <option value="1">First text</option>
          <option value="2">Second text</option>
        </select>
      </div>
   </div>
</div>
```

Scenario 1. Check that the text of the <h5> tag is ‘H5 text’.

The WebElement used for identifying the <h5> tag is:

```
@FindBy(css = "#getTextInnerDiv h5") public WebElement getTexth5;
```

Checking that the text of the <h5> tag is the expected one, by using the ‘getText()’ method is done:

```
assertEquals("H5 text", page.getTexth5.getText());
```

Scenario 2. Check that the text of the link (<a> tag) is ‘Link text’.

The WebElement used for identifying the link is:

```
@FindBy(css = "#getTextInnerDiv a") public WebElement getTextLink;
```

Checking that the label of the link element is the expected one is done as follows:

```
assertEquals("Link text", page.getTextLink.getText());
```

Scenario 3. Check that the button text is ‘Button text’.

The WebElement used for identifying the button is:

```
@FindBy(css = "#getTextInnerDiv button") public WebElement getTextButton;
```

Comparing the expected value of the text on the button with the actual one is done as follows:

```
assertEquals("Button text", page.getTextButton.getText());
```

Scenario 4. Check that the text of the first option from the dropdown is ‘First text’.

The WebElement used for identifying the first option from the dropdown is:

```
@FindBy(css = "#getTextInnerDiv select option") public WebElement
getTextDropdownFirstOption;
```

Comparing the first item text from the dropdown with the expected String is done as follows:

```
assertEquals("First text", page.getTextDropdownFirstOption.getText());
```

Scenario 5. Check that the text of the image is empty.

The WebElement used for identifying the image is:

```
@FindBy(css = "#getTextInnerDiv img") public WebElement getTextImg;
```

Since there is no text displayed on the image, the check the <img> tags' text is an empty String is done as follows:

```
assertEquals("", page.getTextImg.getText());
```

Scenario 6. Check that the text of the tag with id 'getTextInnerDiv' is equal to the sum of texts of all children tags of the tag. These children tags are: h5, img, a, button, select. Ignore any whitespaces.

The WebElement used for identifying the tag is:

```
@FindBy(css = "#getTextInnerDiv") public WebElement getTextInnerDiv;
```

Since the whitespaces are not of interest in the comparison, both the expected and actual texts that will be compared will be stripped off the whitespaces. This way you can check that, apart from any new lines and spaces used for rendering the children elements of the tag, the actual text of the tag (as returned by the 'getText()' method) is composed of the sum of the texts of its' children tags.

```
assertEquals("H5textLinktextButtontextFirsttextSecondtext",
page.getTextInnerDiv.getText().replaceAll("\\s", ""));
```

Scenario 7. Check that the text of the tag with id 'getTextOuterDiv' is equal to the sum of texts of all children tags of the tag. Ignore any whitespaces.

The WebElement used for identifying the tag is:

```
@FindBy(css = "#getTextOuterDiv") public WebElement getTextOuterDiv;
```

The comparison of the expected and actual texts, for this tag, is done in the same fashion as in Scenario 6, by removing any whitespaces from the expected and actual text. The text of the tag is the sum of texts of its' children, which include an <h3> element and the tag described in Scenario 6 (whose text is the sum of texts of all the children of this tag).

```
assertEquals("getText()H5textLinktextButtontextFirsttextSecondtext",
page.getTextOuterDiv.getText().replaceAll("\\s", ""));
```

Scenario 8. Given the following <textarea> element:

```
<textarea maxlength="100">Predefined text</textarea>
```

The corresponding WebElement is:

```
@FindBy(css = "[maxlength='100']") public WebElement getTextTextarea;
```

Calling the 'getText()' method on this WebElement will return the String 'Predefined text', since the definition of this element contains text between the opening and closing tags of the <textarea> element. Calling 'getAttribute("value")' will return the same String. See details on how 'getAttribute()' works in chapter 6.5.

```
assertEquals("Predefined text", page.getTextTextarea.getText());
assertEquals("Predefined text", page.getTextTextarea.getAttribute("value"));
```

Now, if the <textarea> is emptied with the 'clear()' method, the 'getText()' method will still return the initially set text on the tag: "Predefined text". However 'getAttribute("value")' will return an empty String.

```
page.getTextTextarea.clear();
assertEquals("Predefined text", page.getTextTextarea.getText());
assertEquals("", page.getTextTextarea.getAttribute("value"));
```

If you type into the <textarea>, let's say, the text "New text": the result of calling 'getText()' will be as up to now, "Predefined text", whereas the result of calling 'getAttribute("value")' will be, as described in chapter 6.2., the newly typed text ("New text"):

```
page.getTextTextarea.sendKeys("New text");
assertEquals("Predefined text", page.getTextTextarea.getText());
assertEquals("New text", page.getTextTextarea.getAttribute("value"));
```

So, to sum up the behavior of a <textarea>: ‘getText()’ returns the text that was placed between the opening and the closing tag of this element by the developer (in the code) and the result of calling this method will never change, even if you typed something into this field. On the other hand, ‘getAttribute(“value”)’ will return whatever text is visible inside the textfield, whether it was initially placed there by the developer, or whether some typing was done.

GitHub location of example code

- PageObject class, section ‘elements for getText’: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/WebElementInteractionPage.java
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/elementInteractionSolution/ElementInteractionTest.java
- Test methods: getText
- HTML code: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/interactions.html

Chapter References

W3.org. (2019). *HTML 5.2: 4.10. Forms*. [online] Available at: https://www.w3.org/TR/html5/sec-forms.html#the-textarea-element [Accessed 2 Mar. 2019].

W3.org. (2019). *HTML 5.2: 3. Semantics, structure, and APIs of HTML documents*. [online] Available at: https://www.w3.org/TR/html5/dom.html#elements [Accessed 2 Mar. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/ [Accessed 2 Mar. 2019].

6.5. getAttribute()

You already know what an attribute of an HTML element is. Many times in tests you will need to check that certain attributes of the specified HTML elements have the expected values. This is because these attributes define a certain state the element is supposed to have, based on the requirements. If we are talking about an image (the <img> tag), the 'src' attribute tells you what image you are displaying to the customers. For a link element (the <a> tag), the 'href' attribute tells you what URL will open when the customers click on it. For a checkbox or a radio button, the 'checked' attribute signals whether the state of the checkbox/radio button is checked or unchecked.

Whenever you need to check the value of an element's attribute, you will use Seleniums' 'getAttribute()' method. This methods' usage is as follows:

```
webElement.getAttribute(attributeName)
```

Here, the 'webElement' represents the WebElement whose attribute you are interested in, and the 'attributeName' parameter is a String representation of the name of the attribute you are interested in. Remember that the 'getAttribute()' method returns a String value.

One thing worth mentioning is that when you are inspecting a web page in order to see the attributes of an element, you can find attributes in two formats. The first one is where you have an attribute name and an attribute value. This looks like:

```
<elementName attributeName=attributeValue …>
```

Here, obviously the 'getAttribute()' method, called on 'elementName' will return the String value of 'attributeValue'.

The second format is where some attributes do not have explicit values, since they are boolean attributes:

```
<elementName attributeName …>
```

The presence of the attribute in this case means that the value of the attribute is true. Therefore, using the 'getAttribute()' method on such an attribute, when it is present, will return the String 'true'. If the attribute is not present, but you still call the 'getAttribute()' method on this attribute, it will return null (not the String 'null', but an actual null).

The most common boolean attributes you will see on a page you are testing are 'checked' (attached to a checkbox or a radio button) or 'selected' (attached to a dropdown, which a 'select' type of element).

Let us look at some examples and how the 'getAttribute()' method can be used.

6.5.1. Usage example

The following ‘img’ HTML element is given, with an assigned attribute ‘height.’ You will need to check that the value of this attribute is ‘100’:

```
<img id="idModuleImage" src="..."
        width="100"
        height="100"
        alt="This is the module with id">
```

As the first step, in the PageObject class where you define your WebElements, you will need to create a new handle to this element:

```
@FindBy(css = "#idModuleImage")
  public WebElement elementWithId;
```

In the test method, the check for the value of the ‘height’ attribute will be the following:

```
assertEquals("100", page.elementWithId.getAttribute("height"));
```

Notice that in the assertEquals method, you are comparing the value read from the page by the ‘getAttribute()’ method with the String ‘100’, even though when you think of a width or a height, these are expressed as numerical values (hence in Java you would consider them as integer values). This is because the ‘getAttribute()’ method reads the attribute values from the page as a String.

6.5.2. General but not assigned attribute

What happens when you use the ‘getAttribute()’ method on a WebElement, for a general attribute that is not assigned to it? You wouldn’t do this in a test, but it’s good to know what happens if you do, by mistake. Let’s assume the HTML element you will interact with is the following:

```
<img id="idModuleImage" src="..."
        width="100"
```

```
        height="100"
        alt="This is the module with id">
```

As you can see, there is no ‘class’ attribute on this element. The corresponding WebElement will be:

```
@FindBy(css = "#idModuleImage")
   public WebElement elementWithId;
```

The result of calling the ‘getAttribute()’ method on this WebElement, for the ‘class’ attribute, will be an empty String. This can be proven when running the following assertion from a test method:

```
assertEquals("", page.elementWithId.getAttribute("class"));
```

6.5.3. Non-existent attribute

Let’s assume that you need to call the ‘getAttribute()’ method for an elements’ attribute, but the attribute has not been assigned to that element by the developers, by mistake (and this is a bug, as you expected the attribute to be present).

Let’s say the HTML element under test is the following:

```
<img id="idModuleImage" src="..."
        width="100"
        height="100"
        alt="This is the module with id">
```

And that the corresponding WebElement is:

```
@FindBy(css = "#idModuleImage")
   public WebElement elementWithId;
```

If in the test method we have the following assertion, this will pass, since the result of a ‘getAttribute()’ method call for a non existent attribute is a null value:

```
    assertNull(page.elementWithId.getAttribute("nonExistentAttribute"));
```

In other words, if in this case, the assertion would be an ‘assertEquals’ type of assertion, with an expected String value provided as the expected value, the assertion will fail due to the expected attribute not being present.

6.5.4. ‘Class’ attribute composed of multiple values

For the given HTML element, with a ‘class’ attribute:

```
<input class="w3-btn w3-padding w3-border w3-purple" type="button" value="A disabled
button"
        style="font-size:24px;font-family:verdana;"
        disabled>
```

the result of calling the ‘getAttribute()’ method for the ‘class attribute:

```
page.disabledButton.getAttribute("class")
```

will be a String whose value is: “w3-btn w3-padding w3-border w3-purple”.

Therefore, if in your test the requirement is for the element to have just one class that you are interested in, let’s say ‘w3-purple’, the check can be done with an ‘assertTrue’ assertion, where the value obtained with ‘getAttribute()’ contains the expected String:

```
assertTrue(page.disabledButton.getAttribute("class").contains("w3-purple"));
```

If instead you are interested in the ‘class’ attribute to have the exact value "w3-btn w3-padding w3-border w3-purple", use the assertEquals assertion instead:

```
assertEquals("w3-btn w3-padding w3-border w3-purple",
page.disabledButton.getAttribute("class"));
```

6.5.5. ‘Style’ attribute

A rather complex attribute is the ‘style’ one. It is complex because it can hold a lot of information, since its’ purpose is to apply styling to an element inline. An example of such an attribute applied to a button element can be found below:

```
<input class="w3-btn w3-padding w3-border w3-purple" type="button" value="A disabled
button"
        style="font-size:24px;font-family:verdana;"
        disabled>
```

As you can guess, the information from the ‘style’ attribute is related to the font of the element: it will have a size of 24px, and the font family will be Verdana. As you can see, in the above element there are no space characters between the font attributes contained inside the ‘style’ attribute. However, keep in mind that the result of calling the ‘getAttribute()’ method for this attribute will result in a String which is formatted to look more readable: "font-size: 24px; font-family: verdana;". So, if you are interested in both font related styling information to be present in the ‘style’ attribute, you can use the following assertion:

```
assertEquals("font-size: 24px; font-family: verdana;",
page.disabledButton.getAttribute("style"));
```

Or, even better, in order not to to worry about where there are any whitespaces, you can just remove all whitespaces from the expected and actual Strings you are comparing, as in the example below:

```
assertEquals("font-size: 24px; font-family: verdana;".replaceAll("\\s", ""),
page.disabledButton.getAttribute("style").replaceAll("\\s", ""));
```

This way you are just comparing two Strings that have no whitespaces at all.

6.5.6. Boolean attributes

To exemplify using the ‘getAttribute()’ method on boolean attributes, let’s start with checkboxes. Their attribute ‘checked’, when present, signals that the checkbox is checked. The absence of this attribute signals that the checkbox is not checked. So, let’s take the following checkbox HTML element:

```
<input type="checkbox" name="checkedCheckbox" value="chkBx" checked>
```

The corresponding WebElement is:

```
  @FindBy(css = "[name='checkedCheckbox']") public WebElement checkedCheckbox;
```

Let's assume that in the test we require to check that the checkbox is checked. The following assertion will pass successfully when that is the actual behavior:

```
assertEquals("true", page.checkedCheckbox.getAttribute("checked"));
```

For boolean attributes, when they are present, the 'getAttribute()' method returns a String with value 'true'.

Now, for an unchecked checkbox, for which we need to test that is is unchecked, the HTML code will look as follows:

```
<input type="checkbox" name="uncheckedCheckbox" value="unchkBx">
```

Notice there is no 'checked' attribute present on this element. The corresponding WebElement will be:

```
@FindBy(css = "[name='uncheckedCheckbox']") public WebElement uncheckedCheckbox;
```

In the test, we will assert that the result of 'getAttribute()' for the 'checked' attribute is null (since this is a boolean attribute):

```
assertNull(page.uncheckedCheckbox.getAttribute("checked"));
```

Now, let's say we are working with a radio button. When we open the page it is unchecked and we need to make an assertion for this behavior. After the check, we will click on it, and verify that it is checked.

The HTML representation of this radio button when the page is initially open will be:

```
<input type="radio" id="myRadio">The radio button
```

The corresponding WebElement is:

```
@FindBy(css = "#myRadio") public WebElement radioButton;
```

The first check, to verify that the radio button is not checked, will be done by the following assertion:

```
assertNull(page.radioButton.getAttribute("checked"));
```

The radio button will then be clicked, in order to turn it into a ‘checked’ radio button.

```
page.radioButton.click();
```

At this step, the verification that the radio button is checked will be done by the following assertion:

```
assertEquals("true", page.radioButton.getAttribute("checked"));
```

GitHub location of example code

- PageObject class, section ‘iframes page’: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/WebElementInteractionPage.java
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/elementInteractionSolution/ElementInteractionTest.java
- Test methods: getAttribute()
- HTML code: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/interactions.html

Chapter References

W3.org. (2019). *HTML 5.2: 2. Common infrastructure*. [online] Available at: https://www.w3.org/TR/html/infrastructure.html#sec-boolean-attributes [Accessed 15 Feb. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/ [Accessed 15 Feb. 2019].

W3.org. (2019). *HTML 5.2: 3. Semantics, structure, and APIs of HTML documents*. [online] Available at: https://www.w3.org/TR/html/dom.html#elements [Accessed 15 Feb. 2019].

6.6. Select (dropdowns)

In plain HTML, a dropdown is written as a ‘select’ element. A dropdown contains a list of available options that you can select from. Depending on the type of dropdown, you can select one or more options from the available ones: a ‘select’ element which has an attached ‘multiple’ attribute will allow for multiple selection, whereas the lack of the ‘multiple’ attribute means you can only select one value at a time.

The HTML representation of a dropdown with only one possible selection looks along these lines (where ‘...’ represents omitted options):

```
<select name="someName">
  <option value="someValue">someOption</option>
   <option value="someOtherValue">someOtherOption</option>
  ….
</select>
```

The top level tag of a dropdown is the <select> one. This tag can have a series of attributes, including general ones like ‘id’ or ‘class’ for easier identification, or a ‘name’ for the same purpose.

The <select> tag is also the one where the boolean ‘multiple’ attribute will be present, in case this dropdown will be a multiple selection type of dropdown (where ‘...’ represents omitted options):

```
 <select name="someName" multiple>
  <option value="someValue">someOption</option>
…..
</select>
```

Another interesting attribute which is also on the <select> tag is the ‘size’ one, which tells you how many options are visible in the dropdown by default, when you open the page. If you think about a multiple select dropdown, you will usually see more than just one option (without needing to expand the dropdown). This allows for easily choosing more than one option, by holding down the CTRL button while clicking the options you want to select. A dropdown with a ‘size’ attribute set will look something like, where of course ‘someSize’ is a number:

```
 <select name="someName" size="someSize">
  <option value="someValue">someOption</option>
….
</select>
```

Below you can find how to interact with the dropdowns by using methods from the Selenium library. Such interactions include: selecting values from the dropdown in different ways, deselecting, checking the available dropdown options and checking which values are selected.

But before using any of these interaction methods on a dropdown, you need some kind of Java object on which you can use the methods. Up to this point you only used WebElement types of variables for interacting with the HTML elements. For the dropdowns you will work with variables of type 'Select'. You will perform two steps in order to define the variable used for interaction: the first will be creating a WebElement, and the second creating a Select based on the WebElement.

In the PageObject class where you define your WebElements you will need to create a new entry where the WebElement will use the CSS selector of the 'select' element from the HTML:

```
@FindBy(css = "...") public WebElement theSelectWebElement;
```

In order to identify the selector for this WebElement, you need to follow the logic of any other WebElement: look for the unique properties of the 'select' element in the HTML: in case it has a name, use that one. If it has an id, use the id, and so on.

After writing the WebElement, in the test class you will need to create a Select type of variable. Where you create the 'Select' variable is very important: the creation code must be placed after the code which opens the page on which the dropdown can be found, which in turn is after the code that initializes the PageObject. So, let's assume you call the 'PageFactory.initElements()' method in the '@BeforeAll' method. After this step, you open the page which holds the dropdown, but you are doing this still in the '@BeforeAll' method. If you need to use the dropdown reference in several tests, you can place the 'Select' variable creation code in the '@BeforeAll' method too, but after the opening of the page code. This will make the 'Select' available to all the tests that will need to interact with the dropdown. If, instead, you only need to interact with the dropdown in one test, create the 'Select' variable inside that test method.

Creating the 'Select' variable will be done as follows:

```
Select nameOfSelect = new Select(page.theSelectWebElement);
```

Example

For demonstrating the interaction with dropdowns, let's assume the HTML we are working with contains three 'select' tags.

The first one is a single selection type of dropdown, which displays a list of coffee based drinks the customer can choose from:

```
<select name="coffee">
      <option value="1">Espresso</option>
      <option value="2">Doppio</option>
      <option value="3">Cappuccino</option>
      <option value="4">Latte Machiato</option>
      <option value="5">Americano</option>
   </select>
```

As you can see, the ‘select’ tag has a name by which we can identify it and create a WebElement of:

```
@FindBy(css = "[name='coffee']") public WebElement coffeeSelect;
```

In the test class, we will only use this dropdown in one test method, in our example. Therefore, in the test method we will create the corresponding ‘Select’ variable:

```
Select coffeeSelect = new Select(page.coffeeSelect);
```

Another dropdown we will use in our tests will also be a single selection type of dropdown, with a different name on the ‘select’ tag, used for expressing a list of tea based drinks:

```
<select name="tea">
      <option value=""></option>
      <option value="Assam">Assam</option>
      <option value="Ceylon">Ceylon</option>
      <option value="Earl_Grey">Earl Grey</option>
      <option value="Pu_Erh">Pu Erh</option>
      <option value="Lady_Grey">Lady Grey</option>
   </select>
```

The WebElement used for identifying this ‘select’ tag, based on its’ name, is:

```
@FindBy(css = "[name='tea']") public WebElement teaSelect;
```

The corresponding ‘Select’ variable for the tea dropdown is:

```
Select teaSelect = new Select(page.teaSelect);
```

A multiple selection dropdown will also be used to demonstrate the dropdown interactions, and this one will display a list of refreshments the customer can choose from:

```
<select name="refreshment" size="5" multiple>
      <option value="1">Rose Lemonade</option>
      <option value="2">Lavender Lemonade</option>
      <option value="3">Diet Coke</option>
      <option value="4">Still Water</option>
      <option value="5">Sparkling Water</option>
   </select>
```

The WebElement created for this dropdown will use the name for identifying it:

```
 @FindBy(css = "[name='refreshment']") public WebElement refreshmentSelect;
```

The corresponding ‘Select’ variable is:

```
Select refreshmentSelect = new Select(page.refreshmentSelect);
```

There are three dropdowns i will demonstrate, simply because they each have something different. The first one is a dropdown where all the options have a ‘value’ attribute and a text label. The second one has the first option with an empty value and empty text. The third one is a multiple selection dropdown. Also, some of the dropdowns have a ‘text’ ‘value’ attribute for their options, some have a ‘numerical’ ‘value’ (however reading these from the page via Selenium methods will all result in Strings).

6.6.1. Checking the available options

Now that you have a means of interacting with the dropdown, you can check whether the available options in the dropdown are the expected ones. Put in other words: the requirement says that a certain dropdown will contain certain items that the user can choose from. Checking that the dropdown contains the expected options can be done using the ‘getOptions()’ method. A call to this method looks like:

```
nameOfSelect.getOptions()
```

Notice that the call of the method is done from the ‘Select’ variable created for that dropdown. An important thing to remember is that the result of calling this method will be a List of WebElements. Each WebElement from this result represents one ‘option’ tag that can be found as a child of the ‘select’ HTML element. The result of ‘getOptions()’ is therefore a list of all ‘option’ tags that are children of the ‘select’ tag (examples will follows). This method can be applied to dropdowns with one selectable option and with multiple selectable options.

In order to actually check the text displayed inside a dropdown is the expected one, from each of the WebElements obtained via ‘getOptions()’ you need to extract the text using the ‘getText()’ method, add these to a list of actual texts expressed as Strings, and compare this list to the list of expected Strings.

Example

Exemplifying the ‘getOptions()’ method will be done on the dropdowns created in the ‘Example’ subchapter of the previous section, where i described how to create the WebElement and ‘Select’ variable.

Let’s start with the first one, which was the coffee one:

```
<select name="coffee">
      <option value="1">Espresso</option>
      <option value="2">Doppio</option>
      <option value="3">Cappuccino</option>
      <option value="4">Latte Machiato</option>
      <option value="5">Americano</option>
   </select>
```

The ‘texts’ the user can choose from when interacting with this dropdown are: Espresso, Doppio, Cappuccino, Latte Machiato, Americano. Let’s assume we want to write an automated test that checks that this dropdown displays these expected texts. The WebElement and Select variables were already created in the previous section. Now we just need to retrieve the text corresponding to each option displayed in the dropdown, and compare these to the expected values.

We will use the ‘getOptions()’ method. If we used it directly on the ‘Select’ variable, we would obtain a list of WebElements, each corresponding to the ‘option’ tags that are the children of the ‘select’ tag in the HTML. That means: ‘getOptions()’ will not give us the actual text that is displayed in the dropdown for each of the dropdown entries. Therefore we need to do a bit more work. The solution to this task is to go through each of the WebElements resulted from calling the ‘getOptions()’ method, use the ‘getText()’ method on each of these WebElements, add the result of ‘getText()’ to a List of expected Strings, and compare this List to the List of actual Strings.

It might sound complicated but in reality it’s quite simple:

```
List<String> actualCoffeeOptions = new ArrayList<>();
      for (WebElement element : coffeeSelect.getOptions()) {
         actualCoffeeOptions.add(element.getText());
      }
      assertEquals(Arrays.asList("Espresso", "Doppio", "Cappuccino", "Latte Machiato",
"Americano"), actualCoffeeOptions);
```

In the above code, the first line just creates an empty List of Strings, called ‘actualCoffeeOptions’, in which we will store the text read from each ‘option’ tag. In the second line of code, we will go through each WebElement that resulted from calling the ‘getOptions()’ method on the ‘Select’ variable corresponding to the coffee dropdown. Basically we will iterate over all the ‘option’ tags. The third line will just add the text of each ‘option’ tag to the list of actual Strings. The last line of code will just compare the expected list of texts the user will see (which is created inline here, and contains: Espresso, Doppio, Cappuccino, Latte Machiato, Americano) to the actual list of texts, generated via ‘getOptions()’ and ‘getText()’. Of course, in our example, this test will pass.

The second dropdown, the tea one, is a bit more interesting:

```
<select name="tea">
      <option value=""></option>
      <option value="Assam">Assam</option>
      <option value="Ceylon">Ceylon</option>
      <option value="Earl_Grey">Earl Grey</option>
      <option value="Pu_Erh">Pu Erh</option>
      <option value="Lady_Grey">Lady Grey</option>
   </select>
```

As you can see, the first ‘option’ tag has both an empty ‘value’ attribute and an empty label. Visually this means that when the user opens a page where this dropdown is, there will be no text displayed inside the dropdown by default. This is different from what the user sees when looking at the coffee dropdown for the first time: there the ‘Espresso’ text is displayed inside the dropdown. In reality both dropdowns behave the same way: by default, when opening the page and seeing the dropdown for the first time, they will display the text corresponding to the first ‘option’ tag. This comes from the fact that each ‘select’ HTML element must have a ‘size’ set, and if it is not set, a default value is attributed to it. The ‘size’ attribute reflects how many options to display when the dropdown is first rendered. In both these examples, the dropdowns are of single selection type, therefore the default ‘size’ attribute is 1.

Because of the ‘size’ attribute, and the desired behavior of the page the user interacts with, sometimes you want the user to see a pre-selected value in the dropdown. In this case, you will have no ‘option’ with an empty ‘attribute’ tag, just like with the coffee dropdown. However in some situations you don’t want to display a pre-defined text to the user, but instead

display an ‘empty’ dropdown, so as not to influence the user’s choice and to suggest that an interaction with the dropdown is required. Otherwise, in some cases, having a text pre-displayed in the dropdown might lead the user to believe they already selected a value.

When it comes to multiple selection types of dropdowns, there you will either have an explicitly set value for the ‘size’, or, in case it is not specified, it will default to 4.

But coming back to the list of displayed options for the ‘tea’ dropdown, the approach to checking the available texts it will display is identical to the approach for the ‘coffee’ dropdown. The only difference is in the expected values: the first one will be an empty String. It corresponds to the first ‘option’ tag that does not show any text. This test will also pass.

```
List<String> actualTeaOptions = new ArrayList<>();
      for (WebElement element : teaSelect.getOptions()) {
         actualTeaOptions.add(element.getText());
      }
      assertEquals(Arrays.asList("", "Assam", "Ceylon", "Earl Grey", "Pu Erh", "Lady Grey"),
            actualTeaOptions);
```

Checking the available texts the user will see for the multiple selection refreshment dropdown is done exactly the same. The only difference will again be in the list of expected values, which, for the refreshment dropdown will be: Rose Lemonade, Lavender Lemonade, Diet Coke, Still Water, Sparkling Water.

6.6.2. Select by index

The purpose of a dropdown is for the user to choose from one of the options it displays. In Selenium you can use three methods for selecting from a dropdown. The first one is selecting based on index.

When you want to select one option, you are actually selecting one item out of a list. Remember how ‘getOptions()’ returns a List (Java List) of ‘option’ tags. In that List, on position 0 you will have the first option the user sees, on position 1 will be the second item in the dropdown, and so on. Basically, selecting by index means selecting based on the index in the List of available ‘option’ tags.

The method used for selecting by index is ‘selectByIndex()’ and its’ usage is as follows:

```
nameOfSelect.selectByIndex(index)
```

Using this method, you can select one item from the single selection dropdown. If you call this method twice, one after the other, with different indexes as parameters: first you will select the option corresponding to the first index, and then the option corresponding to the

second index. After the code runs, the select option will correspond to the last call to the ‘selectByIndex()’ method. In other words, if your code is in the following order:

```
nameOfSelect.selectByIndex(index1);
nameOfSelect.selectByIndex(index2);
```

After the code runs, the selected option will be that corresponding to index2.

On the other hand, when it comes to the multiple selection dropdown, you can still select an entry using the ‘selectByIndex()’ method. However calling this method several times for different index values is equivalent to performing the multiple selection. This means that if the code above would be run on a multiple selection dropdown, both the options corresponding to index1 and index2 would be selected after the code runs.

Example

In order to select the second item from the coffeeSelect, which has the ‘Doppio’ text, you will use ‘selectByIndex()’ providing a value of 1 for the index:

```
coffeeSelect.selectByIndex(1);
```

6.6.3. Select by value

Another way of selecting an option from a dropdown is by providing the ‘value’ attribute of the desired option to the ‘selectByValue()’ Selenium method. That is, if the options have this attribute. There might be cases when the dropdown you are working with does not have any ‘value’ attribute. For that situation you could use the index based selection method.

In order to select an item from the dropdown, you just need to pass the ‘value’ attribute of the option to be selected to the ‘selectByValue()’ method:

```
nameOfSelect.selectByValue("someValue");
```

Just as in the case of selection by index, multiple selection from a multiple type dropdown is done by calling the ‘selectByValue()’ method multiple times, once for each of the desired values.

Example

Let’s consider the tea dropdown and let’s assume in the test we want to select the ‘Ceylon’ entry. When we inspect the HTML code, we can see that the option corresponding to this entry has a ‘value’ attribute of ‘Ceylon’:

```
<option value="Ceylon">Ceylon</option>
```

Therefore, the selection by value for this tea will be, in the test method:

```
teaSelect.selectByValue("Ceylon");
```

If, let’s say, we wanted to select the Pu Erh tea from the dropdown, we would notice that the corresponding ‘value’ attribute is ‘Pu_Erh’:

```
<option value="Pu_Erh">Pu Erh</option>
```

Therefore, in the test, choosing this option would be done as:

```
teaSelect.selectByValue("Pu_Erh");
```

6.6.4. Select by visible text

The third way of selecting an option from a dropdown with Selenium is based on the text displayed in the dropdown, to the user. This is the text found between each opening <option> tag and the corresponding closing tag. The usage of the ‘selectByVisibleText()’ method is:

```
nameOfSelect.selectByVisibleText(“theVisibleText”)
```

Just as in the case of selection by index and by value, multiple selection from a multiple type dropdown is done by calling the ‘selectByVisibleText()’ method multiple times, once for each of the desired text. Also, just like in the case of selection by index, for this approach you are not interested in whether the option you are selecting has or does not have a ‘value’ attribute, since it is not used when performing the selection.

One thing to keep in mind is that the dropdowns are usually implemented in such a way that the ‘value’ attributes of the options will be constant, whereas the text will be translated, based on how many languages the page supports. This means that there will be just one dropdown implemented, but it will be rendered with different values for the text (found between each opening and closing ‘option’ tag), depending on what language it is rendered in. For this situation, the selection by visible text is not optimal. You would need to create separate tests for each language in which you want to select from the dropdown. For selecting by index or values that is not the case, since the index and values are not changed, no matter what language the page is rendered in.

Example

In this example, from the multiple selection refreshment dropdown, we will select, by visible text, two options: “Still Water” and “Sparkling Water”:

```
refreshmentSelect.selectByVisibleText("Still Water");
refreshmentSelect.selectByVisibleText("Sparkling Water");
```

6.6.5. Checking selected values

After you selected some options from the dropdown, you might want to check that the selection was properly made. You want to make sure that the selected options are in fact those the user chose. But also, for single selection dropdowns, you might want to check that when the page initially loads, the preselected values are the expected ones. As i mentioned earlier, you might want to check that by default an empty option is selected, or maybe one with a desire visible text. For such checks, there are methods that will return either the first or all selected options: ‘getFirstSelectedOption()’ and ‘getAllSelectedOptions()’.

Calling these two methods is done the same way: by applying them to the ‘Select’ variable you created for your dropdown:

```
nameOfSelect.getFirstSelectedOption()
nameOfSelect.getAllSelectedOptions()
```

What is being returned by each method is different: as the name of the method suggests, ‘getFirstSelectedOption()’ returns just one WebElement (corresponding to one <option> tag child of the <select> tag you are working with). On the other hand,

‘getAllSelectedOptions()’ will return a List of WebElements (even if just one option is selected, case in which the size of the List will of course be 1).

When would you use each of these? ‘getFirstSelectedOption()’ is suited for single select type dropdowns, since there will always be just one selected option (as this is how single select dropdowns work). It is not worth calling the ‘getAllSelectedOptions()’ method on the single select dropdowns, since you would always work with a List of size 1, and the purpose of a List is not really to hold just one element. On the other hand, for multiple selection dropdowns, you want to make sure that all options you chose in the test are indeed selected, therefore the ‘getAllSelectedOptions()’ method is more suitable here. However, note that each method can be used for each type of dropdown, but it is just not practical to use them in a different way than i mentioned in this paragraph.

Example

Scenario 1. The page was just loaded and a check needs to be made that the ‘teaSelect’ by default showed the empty option (which is the first option in the <select> element, with an empty text and empty option). The check itself is done as follows:

```
assertEquals("", teaSelect.getFirstSelectedOption().getText());
```

The ‘getFirstSelectedOption()’ method returns the first and only selected <option> tag from the single selection dropdown ‘teaSelect’. In order to compare the text the user sees when this option is select to the expected empty String, the ‘getText()’ method is called on the resulting WebElement from the call of the method ‘getFirstSelectedOption()’.

Scenario 2. After scenario 1 is executed (the page containing the ‘teaSelect’ was open), the option with text “Ceylon” is selected. A check needs to be done on the value of the selected option:

```
teaSelect.selectByValue("Ceylon");
assertEquals("Ceylon", teaSelect.getFirstSelectedOption().getText());
```

Scenario 3. The page containing the ‘refreshmentSelect’ is loaded. The ‘Still Water’ and ‘Sparkling Water’ options are selected. A check needs to be made that the selection, on this multiple selection dropdown, was made successfully:

```
List<String> actualRefreshmentOptions = new ArrayList<>();
refreshmentSelect.selectByVisibleText("Still Water");
refreshmentSelect.selectByVisibleText("Sparkling Water");
```

```
for (WebElement element : refreshmentSelect.getAllSelectedOptions()) {
      actualRefreshmentOptions.add(element.getText());
}
assertEquals(Arrays.asList("Still Water", "Sparkling Water"), actualRefreshmentOptions);
```

Here, the ‘getAllSelectedOptions()’ method returns two WebElements, corresponding to the water beverage <option> tags that were selected from the dropdown. On each of these WebElements, in the ‘for’ loop, a ‘getText()’ method is called, to retrieve the text the user sees in the dropdown, and these Strings are added to a List. In the last line of the previous code a comparison is made between the List of expected texts and the List of the actual texts that are selected in the dropdown.

6.6.6. Deselecting

For multiple selection dropdowns, apart from the methods for selecting from the available options, you also have Selenium methods for deselecting some or all of those options. The usage for these methods, as applied to the desired ‚Select’ variable, is:

```
nameOfSelect.deselectAll();
nameOfSelect.deselectByIndex(index);
nameOfSelect.deselectByValue(value);
nameOfSelect.deselectByVisibleText(visibleText);
```

As the name suggests, the ‘deselectAll()’ method will deselect all the options that were previously selected from a multiple selection type dropdown, and it takes no parameters. The remainder methods are just the ‘deselection’ counterpart methods of the selection methods by index, value and visible text, and each call to such a method will perform a single ‘de-selection’.

As you cannot deselect from a single selection type dropdown, in case in the test you want to select several values from it, one by one, you can just: select the first option you need, do any checks you need, then select the next option you need to work with, and so on. Deselection makes sense for multiple selection type drodpwons, since, let’s say at first you want to have 4 options selected, then another 4. In this case, without deselecting, you could not just select another option, since that will be added to the already set selection.

Example

Scenario 1. For a multiple selection type dropdown, the ‘refreshmentSelect’ one, first open the page and select the water based refreshments (the still and sparkling water). Then, deselect all selected options and check that there are no selected options in the dropdown.

```
refreshmentSelect.selectByVisibleText("Still Water");
refreshmentSelect.selectByVisibleText("Sparkling Water");
refreshmentSelect.deselectAll();
assertEquals(0, refreshmentSelect.getAllSelectedOptions().size());
```

In this code example, first the options are being select, then, by calling the 'deselectAll()' method, all the selected options are being deselected. The easiest check you can do to verify that there is no selected option is to check that the List of WebElements returned by the 'getAllSelectedOptions()' method is of size 0, meaning the List is empty.

Scenario 2. Open the page. From the refreshment dropdown select the options whose text is: Still Water, Sparkling Water, Rose Lemonade. Then, deselect the first option (by index) in the dropdown and check that the only remaining selected options are those whose text is: Still Water, Sparkling Water.

```
refreshmentSelect.selectByVisibleText("Still Water");
refreshmentSelect.selectByVisibleText("Sparkling Water");
refreshmentSelect.selectByVisibleText("Rose Lemonade");
refreshmentSelect.deselectByIndex(0);
actualRefreshmentOptions = new ArrayList<>();
for (WebElement element : refreshmentSelect.getAllSelectedOptions()) {
        actualRefreshmentOptions.add(element.getText());
}
assertEquals(Arrays.asList("Still Water", "Sparkling Water"), actualRefreshmentOptions);
```

Scenario 3. After scenario 2 was executed, deselect (by 'value') the option whose 'value' is '5', and check that the only remaining selected option has the text 'Still Water':

```
refreshmentSelect.deselectByValue("5");
actualRefreshmentOptions = new ArrayList<>();
for (WebElement element : refreshmentSelect.getAllSelectedOptions()) {
    actualRefreshmentOptions.add(element.getText());
}
assertEquals(Arrays.asList("Still Water"), actualRefreshmentOptions);
```

Scenario 4. After scenario 3 was executed, deselect (by visible text) the option whose text is 'Still Water' and check that there are no more selected options in the dropdown.

```
refreshmentSelect.deselectByVisibleText("Still Water");
assertEquals(0, refreshmentSelect.getAllSelectedOptions().size());
```

GitHub location of example code

- PageObject class, section 'dropdowns': https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/WebElementInteractionPage.java
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/elementInteractionSolution/ElementInteractionTest.java
- Test methods: select
- HTML code: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/interactions.html

Chapter References

W3.org. (2019). *HTML 5.2: 4.10. Forms*. [online] Available at: https://www.w3.org/TR/html52/sec-forms.html#the-select-element [Accessed 21 Feb. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/ [Accessed 26 Feb. 2019].

6.7. getCssValue()

HTML elements can be styled, in a sense that properties like colors, size or padding can be set to values imposed by your requirements. Some styling examples include: setting the font family to a desired one, or the font color to a desired one, changing the size of some elements so that they fit properly on the page or in another element, or changing the background color of elements. Styling the HTML elements can be done in several ways, including: having a separate CSS file that changes properties of the elements, or having the CSS code inside the HTML code of the page, setting the 'style' attribute of the elements inline, or using external libraries that will change the expected properties.

Whichever styling property of an element you need to check, you will use Seleniums' 'getCssValue()' method, to which you need to pass the exact name of the property you are checking, and which returns the String value of that property:

```
webElement.getCssValue("propertyName")
```

The best way to understand how this method works, is to go through the following examples.

Example

Scenario 1. The following HTML code is in place (where '...' represents omitted code due to it not being relevant to this example):

```
<div id="getCssValueDiv" ... style="width:300px;">
…
</div>
```

The HTML element is styled inline, through the presence of the 'style' attribute. This attribute sets the width of this element to 300px. The corresponding WebElement is:

```
@FindBy(css = "#getCssValueDiv") public WebElement h2DivElement;
```

Checking that the width is correct is done using the 'getCssValue()' method, by passing the 'width' property (part of the 'style attribute) as parameter to the method. The expected size will be expressed as a String, as will the result of the 'getCssValue()' method call.

```
assertEquals("300px", page.h2DivElement.getCssValue("width"));
```

Scenario 2. In the HTML page where the HTML element under test is defined, in the <head> section, the following <style> tag is defined:

```
  <style>
h2   {color: yellow}
   </style>
```

This means that every <h2> tag on the corresponding HTML page will have a yellow font color. The <h2> element under test is the only one on that page and looks like:

```
<h2>getCSSValue()</h2>
```

The corresponding WebElement is:

```
@FindBy(css = "h2") public WebElement h2Element;
```

The requirement for this element is for its' font color to be '#ffff00'. Many times you will see that the color requirements come just as this one, expressed in hexadecimal codes. This is basically equivalent to the yellow color that was implemented in the HTML, therefore the assertion that the color is '#ffff00' will pass succesfully.

However you cannot simply compare the result of calling 'getCssValue()', on the 'color' property (which gives the font color) to the String "#ffff00". This method, when applied to color properties, like font or background colors, will return a String representation of the 'rgba' color code corresponding to that color, not the 'hex' representation. To make the comparison, you can use Seleniums' Color class. It has a method 'asHex()' that converts the String resulted from 'getCssValue()' to a String representing the hexadecimal color code of that result. This way you will compare two Strings representing hexadecimal color codes.

The test step for checking that the HTML element has the correct yellow font color, using the 'asHex()' method is:

```
assertEquals("#ffff00", Color.fromString(page.h2Element.getCssValue("color")).asHex());
```

Scenario 3. The WebElement under test is styled by using a library called W3CSS. When styling using this library, any element that requires styling has a 'class' attribute whose values are kyewords used by this library to change things like: colors, padding, borders, size, etc. For the purpose of this example, the element under test has a background color in a blue tone, which is set by the 'w3-indigo' value of the 'class' attribute, as follows:

```
<div id="getCssValueDiv" class="w3-indigo ..." ...>
...
</div>
```

In the test you will need to check that there is a correct background color displayed on this element. The WebElement corresponding to this tag is:

```
@FindBy(css = "#getCssValueDiv") public WebElement h2DivElement;
```

The expected color in this case is, again in hexadecimal, '#3f51b5'. If you don't want to convert the result of 'getCssValue()' to a String representing 'hex', you could convert both the expected and actual Strings to a common type: the Color object from Selenium. This means, both the expected 'hex' value and the 'rgba' color returned by 'getCssValue()' will be converted to a Color, so that an expected Color will be compared to an actual Color. In this case you are not interested in what color code is used for either side of the comparison. The 'fromString()' method is used to convert from a String representing a color, in any color code system, to a Color object. You can view this transformation as casting to a common type.

The actual checking of the background color is done as follows:

```
assertEquals(Color.fromString("#3f51b5"),
Color.fromString(page.h2DivElement.getCssValue("background-color")));
```

GitHub location of example code

- PageObject class, section 'for getting the CSS attributes': https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/WebElementInteractionPage.java
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/elementInteractionSolution/ElementInteractionTest.java
- Test methods: getCSSValue
- HTML code: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/interactions.html

Chapter References

Seleniumhq.github.io. (2019). *Support Packages*. [online] Available at: https://seleniumhq.github.io/docs/support.html#working_with_colours [Accessed 3 Mar. 2019].

W3.org. (2019). *Cascading Style Sheets*. [online] Available at: https://www.w3.org/Style/CSS/ [Accessed 3 Mar. 2019].

W3.org. (2019). *Starting with HTML + CSS*. [online] Available at: https://www.w3.org/Style/Examples/011/firstcss [Accessed 3 Mar. 2019].

W3schools.com. (2019). *W3.CSS Home*. [online] Available at: https://www.w3schools.com/w3css/default.asp [Accessed 3 Mar. 2019].

W3.org. (2019). *HTML 5.2: 3. Semantics, structure, and APIs of HTML documents*. [online] Available at: https://www.w3.org/TR/html5/dom.html#the-style-attribute [Accessed 3 Mar. 2019].

Çelik, T. and Etemad, E. (2019). *CSS Style Attributes*. [online] W3.org. Available at: https://www.w3.org/TR/css-style-attr/ [Accessed 3 Mar. 2019].

7. Working with cookies

There are times in your testing when you are interested in the page cookies. Whether you want to delete all of them, in order to have a 'clean' state of the browser, or retrieve some cookie information, you can use Selenium to work with the cookies in your automation code.

First of all, what is a Cookie? It is used by a browser to store some state of the user's activity on a site, and is stored as a small file on the computer where the browser runs. It is saved so that when the user is browsing from one page to another, on the same domain, some needed information on all pages is easily available. Some common cookies are the authentication one, where the logged in users' session is stored, or language cookies. Once the user selects a language on a page, the same language needs to be set across all pages the user accesses, until the language is changed again. Therefore the cookie stores the language selection.

What type of cookie interactions does Selenium allow exactly? Well, you can add cookies to a page, delete a cookie, delete all cookies, and retrieve cookie information (like value or expiry date). What you cannot do however is change a cookie. If, for some reason, you need to change, say the expiry date of the cookie, you will need to first delete that cookie, and add a new one with the properties you need (in this case the same name as the original cookie, the same value, same domain, but different expiry date).

Chapter Example Setup

The test class which contains all the examples of this chapter can be found in GitHub under the following location:
https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/cookies/CookiesTest.java

The import section contains the following entries:

```
import browser.BrowserGetter;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;
import org.openqa.selenium.Cookie;
import org.openqa.selenium.WebDriver;

import java.util.Date;
import java.util.HashSet;
import java.util.Set;
```

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.TestInstance.Lifecycle.PER_CLASS;
```

Apart from the Junit Jupiter imports, which are used for the test annotations (@Test, @BeforeAll, @AfterAll) and the assertions, the BrowserGetter class is also imported, for starting and closing the browser. The WebDriver class is also imported in order to allow for the browser to be interacted with via the 'driver' variable. The Cookie class is imported because adding a cookie to a webpage is done by first creating the Cookie Java object with the properties you are interested in setting, which will be then set as parameter in the 'addCookie()' method. The Set and HashSet classes are imported because the method for retrieving all the cookies from a domain will return a Set, whereas creating an expected Set of cookies will be done through the HashSet (used for comparing the cookies retrieved from the page with Selenium to the expected cookies). The Date class is used for setting the expiry date of a cookie, and comparing the actual expiry date of a cookie retrieved from the webpage under test to the expected date.

This test class will only contain one test method, in which all the relevant cookie related methods will be demonstrated. The test method is what would result if all the example scenarios from this chapter would be performed in sequence in a test. All the cookies setting and deletion in the examples of this chapter will be done on a demonstration site which is freely available to anyone, namely www.example.com, which is open in the @BeforeAll method. That is, right after the driver is initialized. No PageObject class is used in this chapter as no HTNML interaction is required.

7.1. deleteAllCookies()

In order to remove all the cookies that are set when you are visiting via an automated test, you can use Seleniums' 'deleteAllCookies()' method. Its' usage is very straightforward (where 'driver' is the WebDriver instance used in your test):

```
driver.manage().deleteAllCookies();
```

You should check, after calling this method, that the cookies were indeed deleted, by calling the method from subchapter 7.3: 'getCookies()' and checking that the size of the Set resulted from this method call is of size 0 (has no elements).

```
assertEquals(0, driver.manage().getCookies().size());
```

7.2. addCookie()

In order to add a new cookie to the browser, first you need to create a Cookie Java Object. The cookie add method from Selenium does not take Strings as parameters, but instead Cookie type parameters. This Cookie class is also part of the Selenium library. The properties a Cookie object can have are: name, value, path, domain, expiry, isSecure and isHttpOnly. This class has constructors for creating Cookie objects which have at minimum a name and value, and then, depending on the constructor you want to use, you can also set some of the other object properties.

After the Cookie object is created, the 'addCookie()' method will be used, as in the below example, where 'firstCookie' is the name of the Cookie object.

```
driver.manage().addCookie(firstCookie);
```

Note that in order for the cookie to change the behavior of the page, a page refresh is required after adding the cookie. Refreshing the page is discussed in chapter 8, however here is the code for it:

```
driver.navigate().refresh();
```

Example

Scenario 1. Create a new Cookie and set its' name and value properties. Add the cookie to a page.

Creating the Cookie object is done using the class' constructor which takes a name and value as String parameters.

```
Cookie firstCookie = new Cookie("firstCookieName", "firstCookieValue");
```

Adding the cookie to the page is done as follows:

```
driver.manage().addCookie(firstCookie);
```

Scenario 2. Create a new Cookie and set its' name, value and expiry date properties. Add the cookie to the page.

Creating the Cookie object will be done by calling the constructor which takes four parameters: name as String, value as String, the path as String ("/", also known as the root path) and expiry as Date (Java's Date). There is no three parameter constructor corresponding to the requirement for creating the cookie, therefore the constructor which also requires the path to be specified will be used.

```
Cookie secondCookie = new Cookie("secondCookieName", "secondCookieValue", "/",
        new Date(System.currentTimeMillis() + 600000));
```

In this example, the expiry date is set to one minute from the time when the test is run. Adding the cookie to the page can now be done:

```
driver.manage().addCookie(secondCookie);
```

In this case too, in order for the cookie to take effect, a page refresh needs to be performed.

```
driver.navigate().refresh();
```

7.3. getCookies()

When you are interested in checking how many or which cookies are set on a page, you can retrieve a Java Set containing all of them, by using the Selenium 'getCookies()' method. Refer to the 'Chapter References' section for some links where you can read more about Sets. The result of calling this method is a Set of Cookie objects, which means, by iterating over the Set, you can get, for each cookie, all the Cookie object properties (name, value, expiry, and so on), should you need to.

The usage of the 'getCookies()' method is quite simply a call to this method made from the 'driver' variable:

```
driver.manage().getCookies();
```

Now, if you only want to check how many cookies were set on the domain you are working with, you could just check that the size of the resulting Set of Cookie objects is the expected size. It is done by calling the 'size()' method on the resulting Set and comparing the result to the expected size, expressed as ints.

```
assertEquals(expectedSize, driver.manage().getCookies().size());
```

If however you are actually interested in what cookies are set on the page, you could create an expected Set of Cookie objects, which you will compare to the the result of the ‘getCookies()’ method call. Take a look at the examples below

Example

Scenario 3. Given the state of the page after Scenario 1 and Scenario 2 were accomplished (see chapter 7.2.), the current page should now have two cookies set: the ‘firstCookie’ and the ‘secondCookie’. Check that the number of cookies set on the current domain is 2.

Retrieving the cookies set on the page will be done by calling the ‘getCookies()’ method. Checking the number of cookies set on the domain is done by checking the size of the resulting Set of Cookie objects:

```
assertEquals(2, driver.manage().getCookies().size());
```

Scenario 4. Given Scenario 3 is accomplished, check that the cookies set on the current domain are the ‘firstCookie’ and ‘secondCookie’ defined in Scenario 1 and Scenario 2.

An expected Set of Cookie objects must be created, in order to be able to check what cookies are set.

```
Set<Cookie> expectedCookies = new HashSet<>();
expectedCookies.add(firstCookie);
expectedCookies.add(secondCookie);
```

The comparison between the expected and actual Sets of cookies will be done as follows:

```
assertEquals(expectedCookies, driver.manage().getCookies());
```

In case the result of calling the ‘getCookies()’ method is required later in the test, it can be stored to a Set type of variable.

7.4. getCookieNamed()

When you are interested in only one of the cookies set on a domain, you can use the Selenium ‘getCookieNamed()’ method, which will return a Cookie object corresponding to the cookie resulted from this method call (or null if no cookie with the name specified as parameter in this method call exists). The method has the following usage:

```
driver.manage().getCookieNamed("cookieName")
```

The result of calling this method can be stored to a Cookie variable. If you want to check that all this cookies’ properties are the expected ones, you could compare the result of calling the ‘getCookieNamed()’ method to an expected Cookie object.

As an alternative, when you are only interested in one of the cookie properties, you could just compare that property of the Cookie object with the expected value for that property. For each of a Cookie objects’ properties, there is a corresponding getter method, in the Cookie class, for retrieving the value of the property. For example, if you are interested in the ‘value’ of a Cookie variable, you can call the ‘getValue()’ method, which you can then compare to an expected String value.

Example

Scenario 4. For the cookie with name ‘firstCookieName’, set in Scenario 1, check that all the cookie properties are the expected ones.

In Scenario 1 the Cookie Java object named ‘firstCookieName’ was created. In order to check that on the domain under test a cookie corresponding to the ‘firstNameCookie’ Object exists, the comparison will be made between the Cookie Java object and the result of calling the ‘getCookieNamed()’ method. This way all the cookie properties are compared at once (meaning all the properties that were set on the Cookie object). For example, expiry date was not set so it it will not be compared to anything. The comparison will only take into account the ‘name’ and ‘value’ properties, as the Cookie object was only assigned these two properties.

```
assertEquals(firstCookie, driver.manage().getCookieNamed("firstCookieName"));
```

Scenario 5. For the cookie with name ‘secondCookieName’, set in Scenario 2, check that the expiry date is in the future.

In this case, checking that date is in the future means that its’ Date representation is after the Date corresponding to the current time. Therefore this check can be done as follows:

```
int compareToResult = new
Date(System.currentTimeMillis()).compareTo(driver.manage().getCookieNamed("secondCook
ieName").getExpiry());
```

```
assertEquals(-1, compareToResult);
```

The ‘compareTo()’ method from the Date class returns -1 if the Date to the left of the method name in the method call is earlier than the Date specified as the parameter of the method call. Since you are only interested in the expiry date of the cookie, retrieving this value from the page is done by calling the ‘getExpiry()’ getter method on the Cookie object resulted from calling the ‘getCookieName()’ method.

7.5. deleteCookie() and deleteCookieNamed()

In case you want to delete only a specific cookie, you have two options available in Selenium: calling the ‘deleteCookie()’ method which takes a Cookie object as parameter, or calling the ‘deleteCookieNamed()’ method which takes a String parameter. As the names suggest, when you call the ‘deleteCookie()’ method, you need to pass a Cookie object whose properties entirely correspond to the properties of a Cookie stored on the domain where you are running the test. The much simpler way to call a cookie deletion method is by using ‘deleteCookieNamed()’ method, which requires only the name of the cookie to be specified.

The usage of ‘deleteCookie()’, where ‘cookie’ is a Cookie object variable created previously is:

```
driver.manage().deleteCookie(cookie);
```

The usage of ‘deleteCookieName()’, where ‘cookieName’ is the name of the cookie you want to delete is:

```
driver.manage().deleteCookieNamed("cookieName");
```

Example

Scenario 6. Delete the cookie named ‘secondCookieName’ and check that the only remaining cookie is ‘firstCookie’. These were set in Scenario 1 and Scenario 2.

Deleting the ‘secondCookie’ will be done, using the method ‘deleteCookieNamed()’, as follows:

```
driver.manage().deleteCookieNamed("secondCookieName");
```

First create a Set of expected Java objects which in this example will only contain the Cookie ‘firstCookie’.

```
Set<Cookie> expectedRemainingCookie = new HashSet<>();
expectedRemainingCookie.add(firstCookie);
```

Then, just compare the Set of cookies resulting from calling the ‘getCookies()’ method to the Set ‘expectedRemainingCookie’.

```
assertEquals(expectedRemainingCookie, driver.manage().getCookies());
```

Scenario 7. Delete the cookie ‘firstCookie’ and check that no more cookies are set on the current domain.

The ‘secondCookie’ object was created in order to set the cookie on the current domain. The deletion of this cookie will be done by calling the method which takes a Cookie object as parameter, namely ‘secondCookie’. Checking that no more cookies are now set on the page is done by checking that the size of the Set resulting from calling the method ‘getCookies()’ is 0.

```
driver.manage().deleteCookie(firstCookie);
assertEquals(0, driver.manage().getCookies().size());
```

GitHub location of example code

- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/cookies/CookiesTest.java
- Test methods: workingWithCookies

Chapter References

En.wikipedia.org. (2019). *HTTP cookie*. [online] Available at: https://en.wikipedia.org/wiki/HTTP_cookie [Accessed 5 Mar. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: fhttps://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/Cookie.html [Accessed 5 Mar. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/WebDriver.Options.html [Accessed 5 Mar. 2019].

Docs.oracle.com. (2019). *Date (Java SE 10 & JDK 10)*. [online] Available at: https://docs.oracle.com/javase/10/docs/api/java/util/Date.html [Accessed 5 Mar. 2019].

Docs.oracle.com. (2019). *Set (Java SE 10 & JDK 10)*. [online] Available at: https://docs.oracle.com/javase/10/docs/api/java/util/Set.html [Accessed 6 Mar. 2019].

Docs.oracle.com. (2019). *The Set Interface (The Java™ Tutorials > Collections > Interfaces)*. [online] Available at: https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html [Accessed 6 Mar. 2019].

8. Page Navigation

In the previous chapter, in order for a cookie that you set on a page to take effect, a page refresh is required. In other tests you might need to go back to the previously open page, just as the user would in a browser by clicking the 'Back' button. For such cases, Seleniums' navigation methods can be used. They help refresh the page, go back to the previous page, and go forward to the next page (if it was previously already visited during the test, just like in the browser, where if you are at the last page of the navigation history the forward button is not active).

The usage of these methods is straightforward. Given that the 'driver' variable holds the WebDriver instance, refreshing the page is done as follows:

```
driver.navigate().refresh();
```

Similarly, going back to the previous page is done as follows:

```
driver.navigate().back();
```

And navigating forward:

```
driver.navigate().forward();
```

Some checks might be needed, in order to, for example, check whether after navigating in the required direction, the correct URL has loaded. For this check the 'getCurrentUrl()' method can be used. Depending on the requirements, other checks might also be needed.

One thing you need to consider when using these methods is that in some cases the result of navigating back and forth will not be what you expect. Depending on the system on which you are testing, getting from one page to the other by clicking on buttons or submitting

forms may be performed by means of some ‘invisible’ redirects. I say ‘invisible’ because when you look at the browser during testing, you don’t see that between the page you are clicking a button on and the resulting page some other URLs are loaded in the browser for a fraction of a second. Basically, when submitting a form, some additional processing is done, which requires the forms to be submitted to a URL that processes the data submitted and then redirects either to another intermediate page or to the page the user needs to see. Those intermediate pages do not contain anything useful to the user, and many times they are just empty pages. But the fact that these redirects exist, might make nagivating back and forward impredictible, by landing you on intermediate pages you might not be aware of. Therefore, you need to be aware of how the pages you are testing are implemented, in order to understand whether navigation back and forward in your tests is really feasible.

Chapter Example Setup

The test class used for demonstrating the navigation methods is located at: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/navigation/NavigationTest.java .

The import section contains the following entries:

```
import browser.BrowserGetter;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.PageFactory;
import tutorialSolution.pages.WebElementInteractionPage;

import java.io.File;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;
```

These are imports seen in other chapters too, which cover JUnit related classes, the BrowserGetter class for starting the browser, the PageFactory class used for instantiating the WebElements from the WebElementInteractionPage class (since in the tests you will click on a link from this page), and the WebDriver class used for keeping the driver instance and for calling the navigation methods used in the examples below.

The @BeforeAll method will initialize the browser and the PageObject class, whereas in the @AfterAll method the browser will be closed.

```
@BeforeAll
```

```
    public void beforeAll() {
        driver = browserGetter.getChromeDriver();
        page = PageFactory.initElements(driver, WebElementInteractionPage.class);
}

@AfterAll
public void afterAll() {
        driver.quit();
}
```

The URL on which the link that needs to be clicked can be found is open inside the test method (called ‘navigation()’), since there is only one test method in the entire test class.

Example

Scenario 1. Open a page and click on a link which will cause a different URL to be loaded. Go back to the previous page and check that the URL loaded in the browser is that of the page initially opened.

The second url which will be open (by clicking the link on the first page) will be defined in a String variable:

```
String redirectUrl = "https://www.example.com/";
```

In this example, the ‘interactions.html’ page will be open, which was also used in previous examples. First a check is made, by calling the ‘getCurrentUrl()’ method, that the url that has loaded in the browser is the correct one, corresponding to the ‘interactions.html’ page. The result of this ‘getCurrentUrl()’ method will be stored in the ‘currentUrl’ variable, to be used for comparison later.

```
driver.get(new File("src/main/resources/interactions.html").getAbsolutePath());
String currentUrl = driver.getCurrentUrl();
assertTrue(currentUrl.contains("interactions.html"));
```

Then, the link which makes the desired redirect (represented by the ‘linkToClick’ WebElement variable) will be clicked. A check will be made that clicking the link will load the expected redirect URL in the browser (defined in the ‘redirectUrl’ variable):

```
page.linkToClick.click();
assertEquals(redirectUrl, driver.getCurrentUrl());
```

Now, as per the requirements of Scenario 1, the ‘back()’ method will be used to navigate from the currently loaded URL in the browser back to the URL that was initially loaded. A check will be made using ‘getCurrentUrl()’.

```
driver.navigate().back();
assertEquals(currentUrl, driver.getCurrentUrl());
```

Scenario 2. Go forward and check that the URL loaded in the browser is of the page where you were redirected when clicking on the link you clicked at Scenario 1.

Going forward will be done by using the ‘forward()’ method and the check that the correct URL was loaded is done again with the ‘getCurrentUrl()’ method:

```
driver.navigate().forward();
assertEquals(redirectUrl, driver.getCurrentUrl());
```

Scenario 3. Refresh the page and check that the URL is the same as before the page refresh.

This is done by using the ‘refresh()’ method.

```
driver.navigate().refresh();
assertEquals(redirectUrl, driver.getCurrentUrl());
```

GitHub location of example code

- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/navigation/NavigationTest.java
- Test methods: navigation

Chapter References

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/WebDriver.Navigation.html [Accessed 7 Mar. 2019].

W3.org. (2019). *HTML 5.2: 4.10. Forms*. [online] Available at: https://www.w3.org/TR/html52/sec-forms.html#sec-forms [Accessed 7 Mar. 2019].

9. Working with windows/tabs

Up to now, in all the previous tests, there was only one open window or tab in which the work was performed. When testing requires you to work with multiple open windows or tabs, Selenium is here to help. A new window or tab usually opens when a user clicks on a button or link which triggers the new page that loads to be open in a new window or tab. Whether it is a window or tab that opens depends on the browser you are using. No matter whether you have a new tab or a new window that opens, the methods presented in this chapter for working with them behave exactly the same. They make no distinctions in regards to what was open (a tab or window).

How can you tell that the page that opens when clicking on a link opens in a new window/tab? By inspecting the links' 'target' attribute. If the value of this attribute is '_blank', than the page that will open will be inside a new window/tab. If there is no 'target' specified, or its' values are different than '_blank', no new window/tab will be open when clicking on this link.

The methods that Selenium provides when working with windows/tabs are as follows: 'getWindowHandles()' which returns handles to all currently open windows/tabs (windows which were opened through the driver instance); 'getWindowHandle()' which gets the handle to the window in focus (the current window you are working with); 'window()' used as 'driver.switchTo().window(windowHandle)' which allows to bring another window into focus, in order to interact with the page opened in that window.

9.1. getWindowHandles()

When you have several open windows, in order to switch from one to the other to allow for page interaction, you will need to have a way of identifying the windows you are trying to switch to. Each window or tab has a value associated to it, called a handle. The handle is nothing more than a unique set of characters that identifies it, like a label. The result of calling the 'getWindowHandles()' method from Selenium is: all the handles of the open windows as a Java Set of String values. Calling this method from a test can be done in the following manner:

```
driver.getWindowHandles()
```

If you wish to store the result of calling this method, in order to access the handles later in a test, you can use a variable of type Set of String values:

```
Set<String> setOfWindowHandles = driver.getWindowHandles();
```

Using the handles returned by this method in tests is exemplified in the Examples section of this chapter.

One very important aspect to mention is that the window handles returned by the ‘getWindowHandles()’ are only of those windows opened by the automated tests you are running. As you already know, in order to open a page in a brower, you must first initialize (or start) the browser. When you are doing this, you are actually initializing a ‘driver’ instance. Therefore any new page which opens while you are interacting with this driver instance will also be related to the driver instance. Therefore, the ‘getWindowHandles()’ method only returns driver related window handles. Any other browser windows that are open independent of the ‘driver; are not returned by this method. For example, if before running a test which opens a browser, you already had some browser windows open, their handles are not returned by this method.

9.2. getWindowHandle()

If you need to get a handle to the currently focused window, in order to switch back to it later in a test, without also retrieving the handles of any other open page, you can use the ‘getWindowHandle()’ method. It will return the handle to the currently focused driver instance window as a String.

```
driver.getWindowHandle()
```

As you will need this handle later on in the test (that is why you would call this method in the first place), you can store to it to a String variable:

```
String currentWindowHandle = driver.getWindowHandle();
```

9.3. switchTo().window()

Now that you have a way of identifying the open driver instance windows/tabs, they are useful for being able to switch the focus to the desired ones. In order to switch to a window, if you already have its handle saved in variable (as per 9.2. where ‘getWindowHandle()’ was used to store a single window handle), you can just pass the variable as parameter to the ‘window()’ method as follows:

```
driver.switchTo().window(handle);
```

In case you stored all the window handles in a Set (as per 9.1. where ‘getWindowHandles()’ was used), you will need to iterate through the Set, in order to get to the handle you need, and use that handle as parameter to calling the ‘window()’ method. See Chapter Examples section.

An important thing to remember is that when you are providing a non existent value for the handle, as parameter to the ‘window()’ method, you will get a ‘NoSuchWindowException’ exception. This means that you either got the handle wrong (you manually typed it for some reason and did not type the value of an existent handle), or the handle is not valid anymore (the corresponding window was closed between the time the handle was generated and the time when you are trying to switch to it).

Example Setup

The tests from this example will be performed on the same setup: a main page will be opened, named ‘mainePage.html’ (which you can find in the code project at the following location: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/mainPage.html). On this main page, there are two links (<a> tags). One of them opens a second page in a new window/tab, whose name is ‘secondPage.html’ (the HTML code for this page can be found at the following location: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/secondPage.html). The other link opens a third page, with the name ‘thirdPage.html’ (which can be found in the following location: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/thirdPage.html)

The relevant part of the HTML code corresponding to the first page (which i will refer to as the ‘main page’ in this chapter) is:

```
<body>
<header>
   <div class="w3-padding-small w3-center w3-card-4 w3-blue">
      <h1>Welcome to the main page</h1>
   </div>
</header>

<div class="w3-card-4 w3-margin w3-padding-small w3-center" >
   <h3>This is the first page</h3>
   <a href="secondPage.html" target="_blank">Go to second page</a>
   <a href="thirdPage.html" target="_blank">Go to third page</a>
</div>
</body>
```

As you can see, there are two <a> tags, which both have a ‘target’ attribute with the ‘_blank’ value. Therefore, when any of these links is clicked, the corresponding page (specified as the ‘href’ attribute) will be open in a new window/tab. There is also an <h1> tag with a text indicating that this is the main page.

The relevant part of the HTML code corresponding to the second page is:

```
<body>
<header>
   <div class="w3-padding-small w3-center w3-card-4 w3-sand">
      <h1>Welcome to the second page</h1>
   </div>
</header>

<div class="w3-card-4 w3-margin w3-padding-small w3-center" >
   <h4>This is the second page</h4>
</div>
</body>
```

This page also has an <h1> tag whose text indicates that this is the second page.

The relevant part of the HTML code corresponding to the third page is:

```
<body>
<header>
   <div class="w3-padding-small w3-center w3-card-4 w3-sand">
      <h1>Welcome to the second page</h1>
   </div>
</header>

<div class="w3-card-4 w3-margin w3-padding-small w3-center" >
   <h4>This is the second page</h4>
</div>
</body>
```

On this page you can also see an <h1> tag, whose text indicates that this is the third page.

A new PageObject class will be defined for the tests in this chapter, called ‘WindowsPage’. It will only have three WebElements: the two link elements and another one corresponding to the <h1> tag which is present on all three pages that the tests will go through.

```
@FindBy(css = "[href*='secondPage']") public WebElement linkToSecondPage;
@FindBy(css = "[href*='thirdPage']") public WebElement linkToThirdPage;
@FindBy(css = "h1") public WebElement h1Element;
```

There will not be one separate WebElement for each <h1> tag from each page, but instead only one such WebElement. By switching windows, and, for example, using the 'getText()' method on the WebElement corresponding to the <h1> tag, you will get the text of the <h1> tag present on the page loaded in the currently selected window (the one which is in focus). When switching to a different window, whose loaded page also contains an <h1> tag, the 'getText()' method called on this tag will return the text of the <h1> tag corresponding to the current page.

Inside the test class, 'WindowsTest', the before and after methods are now: @BeforeEach and @AfterEach. This means that the browser will be open before each test method runs, and it will be closed after each test method runs. This is to keep the state of the browser clean for each test start, as each test will open several windows/tabs. It is easiest to cleanup after a test which opens multiple windows/tabs by calling the 'quit()' method after the test run, which closes all the open browser instances (spawned as driver instances). Also, in the @BeforeEach method, the maine page is open. From this page new pages (in new windows/tabs) will be open in each test.

Chapter Examples

Scenario 1:

Step 1. From the main page, save the window handle (in order to switch to this window in a later step).

Step 2. From the main page, click on both links. Check that there are 3 open windows.

Step 3. Check that the second page opened properly in one of the open windows (by checking that the url of one of the open windows contains the name of the page, which is 'secondPage'). Check that on the second page the <h1> tag displays the following text: 'Welcome to the second page'.

Step 4. Switch to the main page window and check that the switch was made correctly.

Solution:

Step 1. As the @BeforeEach method was run, the main page was opened in the currently only open window. Therefore, storing the handle of the currently open window to a variable is done by calling the 'getWindowHandle()' method:

```
String mainPageWindowHandle = driver.getWindowHandle();
```

Step 2. From the maine page, which is loaded in the main window, click on the two links. This will bring up two more window/tabs.

```
page.linkToSecondPage.click();
page.linkToThirdPage.click();
```

Now the check that there are 3 open windows/tabs should be done. In order to have something to count, first the handles of all open windows (through the driver instance) will be stored to a variable called ‘setOfWindowHandles’, which is a Set of String values.

```
Set<String> setOfWindowHandles = driver.getWindowHandles();
```

In order to check that there are 3 open windows/tabs, we need to check that there are 3 window handles. That means we will check the size of the ‘setOfWindowHandles’ Set.

```
assertEquals(3, setOfWindowHandles.size());
```

Step 3. In order to check that the second page (whose name contains the text ‘secondPage’) has loaded in a new window/tab, an iteration through the elements of the Set (the handles) will be made, with a ‘for’ loop. Before the iteration is made, a new boolean variable, called ‘foundWindow’ is created and set to ‘false’. You will see why in a second.

As there is the need to check that the second page is loaded in a new window/tab and also that the <h1> tag on that page needs to have a specified expected text, the iteration will be made as follows: all the existing window handles will be iterated over; if the current window over which is being iterated has a URL containing the text ‘secondPage’, that means that the second page is indeed open in this window; in this case the subsequent check, for the <h1> tag text is made. However, if no window with the expected url is found, without the boolean variable created before the iteration, there would be no way to realize that the window is not present. Therefore, if the window is found in the iteration phase, the boolean is set to true, which means that the window was found. If the window is not found, the boolean’s value is not changed and remains false. After the iteration is done, a check is made that the boolean value was set to true.

If this boolean value would not have been used, within the iterator, if the page was not found, the iterator would simply go through all the window handles, and if no handle whose window was displaying the second page is found, the for loop would just exit successfully.

Therefore, the check for second page to be loaded in a new window/tab, together with the check of the <h1> tag text, is as follows:

```
boolean foundWindow = false;
      for (String handle : setOfWindowHandles) {
         driver.switchTo().window(handle);
         if (driver.getCurrentUrl().contains("secondPage")) {
            assertEquals("Welcome to the second page", page.h1Element.getText());
            foundWindow = true;
         }
      }
assertTrue(foundWindow, "No such window was open");
```

Step 4. Now, switch to the window whose handle was saved into the ‘mainPageWindowHandle’ variable in the first step of this test scenario. Check that the main page is in focus. This is done by checking that, after switching to the window whose handle is ‘mainPageWindowHandle’, the URL of the window which is now in focus is that of the ‘mainPage’.

```
driver.switchTo().window(mainPageWindowHandle);
assertTrue(driver.getCurrentUrl().contains("mainPage"));
```

Scenario 2:

Step 1. From the main page, click the two links. Check that there are now three open windows.
Step 2. Close only the window which displays the second page. Check that now only 2 windows are open.
Step 3. Switch to the main window. Click on the link which opens the second page again. Check that there are now 3 open windows.

Solution:

Step 1. Clicking on the links from the main page is rather simple:

```
page.linkToSecondPage.click();
page.linkToThirdPage.click();
```

In order to check that there are currently three open windows, the size of the result of calling the ‘getWindowHandles()’ method is checked to be 3.

```
assertEquals(3, driver.getWindowHandles().size());
```

Step 2. Now, switch to the window which displays the second page. Since the order of window handles is random, an iteration will be made through the Set of window handles, and when the window whose current url contains ‘secondPage’ (corresponding to the second page), it will be closed with the ‘close()’ method.

```
Set<String> setOfWindowHandles = driver.getWindowHandles();
      for (String handle : setOfWindowHandles) {
         driver.switchTo().window(handle);
         if (driver.getCurrentUrl().contains("secondPage")) {
            driver.close();
         }
      }
```

Now, in order to check the number of open windows, the number of existing window handles needs to be compared to 2.

```
assertEquals(2, driver.getWindowHandles().size());
```

In the previous code, a variable of type Set of String was created, to store the handles to the open windows. This Set is not relevant anymore, after one of the windows was closed. This is because the window handle corresponding to the window that was closed does not exist anymore. Therefore, trying to switch to the window handle that was previously assigned to the window that was just closed, will lead to an exception of type ‘NoSuchWindowException’. In order for that set to contain relevant and accurate information, it should be regenerated, by calling the ‘getWindowHandles()’ method again and storing the result of this metho into the Set variable.

```
setOfWindowHandles = driver.getWindowHandles();
```

Step 3. Now, a switch to the main page window needs to be done. In this scenario, the handle of the main page window was not stored into a variable at the beginning (as in Scenario 1), to demonstrate another way of switching to the main page window. In this approach, an iteration with a ‘for’ loop is made over the currently open window handles, by switching to each window. When the main window page is encountered (the current url of the window in focus is checked), the ‘for’ loop will be exited with the ‘break’ option. That means that at the time the ‘break’ is encountered, the currently selected window is that of the maine page.

```
for (String handle : setOfWindowHandles) {
        driver.switchTo().window(handle);
        if (driver.getCurrentUrl().contains("mainPage")) {
            break;
        }
    }
```

Now, from the main window, the link to the second page is clicked. Then, the check that the number of existing window handles is 3 is done, without regenerating the Set of currently open windows. This is because this Set will not be used in the test anymore, as the check that there are 3 open windows is the last step of this step. Otherwise, if other test steps would be required that would interact with different open windows, the Set of windows handles would need to be regenerated.

```
page.linkToSecondPage.click();
assertEquals(3, driver.getWindowHandles().size());
```

GitHub location of example code

- PageObject class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialSolution/pages/WindowsPage.java
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialSolution/windows/WindowsTest.java
- Test methods: savingHandlesToSet, closeOneOfOpenWindows
- HTML code: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/mainPage.html https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/secondPage.html https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/thirdPage.html

Chapter references

W3.org. (2019). *HTML 5.2: 6. Loading Web pages*. [online] Available at: https://www.w3.org/TR/html5/browsers.html#browsing-context-names [Accessed 17 Mar. 2019].

Seleniumhq.github.io. (2019). *WebDriver*. [online] Available at: https://seleniumhq.github.io/docs/wd.html#webdriver [Accessed 17 Mar. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/WebDriver.TargetLocator.html [Accessed 17 Mar. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/WebDriver.html [Accessed 17 Mar. 2019].

Seleniumhq.github.io. (2019). *Generated Documentation (Untitled)*. [online] Available at: https://seleniumhq.github.io/selenium/docs/api/java/index.html?org/openqa/selenium/NoSuchWindowException.html [Accessed 17 Mar. 2019].

10. User prompts

In your tests you might encounter specialized popups, which are generated via Javascript, and which are called ‘user prompts’. These are very basic in functionality, and they come in three variants: an ‘alert’ which only displays an informational message and an ‘OK’ button; a ‘confirm’ which displays an informational message, together with an ‘OK’ and ‘Cancel’ button; a ‘prompt’ which displays an informational message, possibly an input field for typing, and an ‘OK’ and ‘Cancel’ button.

Such a user prompt will be displayed usually when a user clicks a button, which triggers the prompt to be displayed. Understanding what type of user prompt will be displayed when a user clicks a button can be done by inspecting the HTML code of the button. It will have an attribute called ‘onclick’, whose value is the name of a Javascript function that will execute when the button is clicked. The location of the function can be either in the HTML file itself, or in a different file which is imported in the HTML file. You will find this information in the <script> section of your HTML file. If you find the function with the name equal to the ‘onclick’ attribute value, there you have it. If not, you will find an ‘import’ declaration, such as: <script src=”someExternalJsFile”>, where “someExternalJsFile” is the name of the file where the function which triggers a user prompt is defined.

The function you are looking for is of the following format:

```
<script>
function nameOfFunction() {
   ...
}
</script>
```

Of course, a script which triggers a user prompt to be displayed might do more than just show you the prompt. Based on what you click, some additional code might be executed in the script. That will depend on what product you are working on. However, the code which simply displays the user prompts is as follows:

- Alert: this is a simple prompt with a text displayed (“theAlertMessage” in this example) and a OK button:

```
alert("theAlertMessage");
```

- Confirm: this is a prompt with a text displayed (“theConfirmMessage” in this example) and an OK and Cancel buttons

```
confirm("theConfirmMessage");
```

- Prompt:
    - this is a prompt with a text displayed (“thePromptMessage” in this example), an empty input field in which the user can type, and an OK and Cancel buttons:

```
prompt("thePromptMessage");
```

    - this is a prompt with a text displayed (“thePromptMessage” in this example), a pre-filled input field (with the text “thePromptInputField”) in which the user can type, and an OK and Cancel buttons:

```
prompt("thePromptMessage", “thePromptInputField”);
```

Chapter Example Setup

The HTML page that will be used in this chapter contains three buttons. Clicking on each of them will trigger a different type of user prompt to be displayed.

```
……
<button id="alertButton" onclick="alertFunction()">Alert button</button>
<button id="confirmButton" onclick="confirmFunction()">Confirm button</button>
<button id="promptButton" onclick="promptFunction()">Prompt button</button>

<p id="userClicked"></p>

<script>
function alertFunction() {
  alert("This is the alert box");
}
function confirmFunction() {
  var confirmBoolean = confirm("Confirm modal");
  if (confirmBoolean == true) {
    document.getElementById("userClicked").innerHTML = "OK";
  } else {
    document.getElementById("userClicked").innerHTML = "Cancel";
  }
   txt;
}
function promptFunction() {
  var promptOption = prompt("Type something");
```

```
  if (promptOption != null) {
    document.getElementById("userClicked").innerHTML =
    "OK";
  }
  else {
  document.getElementById("userClicked").innerHTML =
    "Cancel";
  }
}
</script>
……..
```

After this page is opened in the browser, when the button with id ‘alertButton’ is clicked, the ‘alertFunction()’ function from the <script> section of this page is executed. It triggers an ‘alert’ to be displayed. This alert displays the ‘This is the alert box’ text, and it also has an ‘OK’ button.

When the button with id ‘confirmButton’ is clicked, the ‘confirmFunction()’ function from the <script> section of this page is executed. It triggers a ‘confirm’ to be displayed. This confirm displays the ‘Confirm modal’ text, and it also has an ‘OK’ and a ‘Cancel’ buttons. When the user clicks on any of the two buttons, in the <p> tag with id ‘userClicked’ a text is displayed: ‘OK’ if the user clicked ‘OK’ and ‘Cancel’ if the user clicked the other button.

When the button with id ‘promptButton’ is clicked, the ‘promptFunction()’ function from the <script> section of this page is executed. It triggers a ‘prompt’ to be displayed. This prompt displays the ‘Type something’ text, has a typable input field, and it also has an ‘OK’ and a ‘Cancel’ buttons. When the user clicks on any of the two buttons, in the <p> tag with id ‘userClicked’ a text is displayed: ‘OK’ if the user clicked ‘OK’ and ‘Cancel’ if the user clicked the other button.

Note: the paragraph which displays what the user clicked is created in this example only for demonstration purposes. It is up to the application you are developing to add some specific behavior based on what the user clicks on inside the ‘prompt’ or ‘confirm’.

The import section of the test (https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialsolution/userprompts/UserPromptsTest.java) will contain the following entries:

```
import browser.BrowserGetter;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;
import org.openqa.selenium.Alert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.PageFactory;
import tutorialsolution.pages.UserPromptsPage;
```

```
import java.io.File;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.TestInstance.Lifecycle.PER_CLASS;
```

As in previous tests, the junit jupiter related import are used for the setup methods, for the test methods and for the assertion parts. The BrowserGetter, WebDriver and PageFactory imports are used for starting the browser, storing the driver instance and initializing the PageObject class elements. The UserPromptsPage is imported because that is the PageObject class where the WebElements for this test are created. The relevant import for demonstrating user prompts is the Alert class.

The @BeforeAll method contains the browser and PageObject initialization, and the code which opens the ‘userPrompts.html’ page, where the code for the user prompt demo is stored. The browser will be closed in the @AfterAll class.

```
@BeforeAll
    public void beforeAll() {
        //initialize the Chrome browser here
        driver = browserGetter.getChromeDriver();
        //initialize page object class
        page = PageFactory.initElements(driver, UserPromptsPage.class);
        driver.get(new File("src/main/resources/userPrompts.html").getAbsolutePath());
    }

    @AfterAll
    public void afterAll() {
        driver.quit();
    }
```

The PageObject class (https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialsolution/pages/UserPromptsPage.java) contains the WebElements corresponding to the buttons which, when clicked, trigger the alert/confirm/prompts. It also contains WebElement of the paragraph in which, when the user clicks a button on confirm/prompt, a text corresponding to which button was clicked is displayed:

```
@FindBy(css = "#alertButton") public WebElement alertButton;
@FindBy(css = "#confirmButton") public WebElement confirmButton;
@FindBy(css = "#promptButton") public WebElement promptButton;
@FindBy(css = "#userClicked") public WebElement userClicked;
```

10.1. Alert

The alert type of user prompt:

- Displays:
  - A text
  - An OK button
- Has the following JS code: **alert**("theAlertMessage");
- Can be interacted with from Selenium by:
  - Clicking the OK button
  - Reading its text

In case more than one interaction with the alert is made in a test, it is worth storing a handle to the alert by creating an ‘Alert’ object variable. Getting the handle to it is easily done in the following code:

```
Alert alert = driver.switchTo().alert();
```

Now, in order to read the text of the alert or to click the OK button, while the alert is displayed, you can call the relevant methods for these actions directly on the ‘alert’ variable. Reading the label of the alert is done by using the ‘getText()’ method discussed in Chapter 6:

```
alert.getText()
```

Clicking on the OK button of this alert is easily done by using the ‘accept()’ method on the ‘alert’ variable:

```
alert.accept()
```

Once you defined the variable, you don’t need to call the ‘driver.switchTo()’ method, if you closed the alert and opened it again. Once the variable is defined for an alert, you can use it across your test, but only for that particular alert, as it returns a handle to that alert.

If you only need to perform one action on the alert in your test (maybe just closing it), you don’t need to create a separate variable for the alert. It is not worth creating a variable which will only be used once. Therefore, in such a case, you could easily close the alert by writting the following code in your test, where you are chaining the method calls (first you switch to the alert, then click on it):

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

This approach also has the benefit of writing less lines of code.

Note: when using the alert related method, like ‘accept()’ or ‘getText()’, you need to make sure you are doing this while the user prompt, which ever it is (alert,confirm or prompt) is displayed. If you try to use these methods when the alert is not displayed you will encounter the following ‘NoAlertPresentException’ Exception:

```
org.openqa.selenium.NoAlertPresentException: no alert open
```

Example

Scenario 1. Click on the button which opens an alert. Check that the text of the alert is ‘This is the alert box’. Then, close the alert.

Clicking on the button which will trigger the alert to be displayed is straightforward (see Chapter Example Setup section):

```
page.alertButton.click();
```

Now that the alert is displayed, create a new variable which will store a handle to the alert, so that you can interact with the alert in the tests:

```
Alert alert = driver.switchTo().alert();
```

This variable is of type ‘Alert’. Now you can read the text of the alert using the ‘getText()’ method and compare it to the expected one.

```
assertEquals("This is the alert box", alert.getText());
```

The last step of this test requires closing the alert, by using the ‘accept()’ method:

```
alert.accept();
```

Scenario 2. Click on the button which opens an alert. Close the alert.

For this test, there is only one action that needs to be performed on the alert, namely closing it once it is open. Therefore, there is no need to store the handle to the alert into a new

variable. Instead, the entire test will consist of two steps: clicking the button which triggers the alert, then closing the alert:

```
page.alertButton.click();
driver.switchTo().alert().accept();
```

10.2. Confirm

The confirm type of user prompt:

- Displays:
  - A text
  - An OK button
  - A Cancel button
- Has the following JS code: **confirm**("theConfirmMessage");
- Can be interacted with from Selenium by:
  - Clicking the OK button (accept)
  - Clicking the Cancel button (dismiss)
  - Reading its text

In case more than one interaction with the alert is made in a test, it is worth storing a handle to the confirm by creating an ‘Alert’ object variable. From a Selenium point view, whether the user prompt is an alert, a confirm or a prompt, the handle to a user prompt is stored in an ‘Alert’ object variable (just like in subchapter 10.1). There is no ‘Confirm’ or ‘Prompt’ variable.

Getting the handle to it is easily done in the following code, which is also identical to the way you would get the handle to an alert (as in subchapter 10.1) or to a prompt:

```
Alert alert = driver.switchTo().alert();
```

Retrieving the text of the ‘confirm’ is also done just as for the alert from the previous subchapter:

```
alert.getText()
```

It is the same with clicking on the ‘OK’ button of the prompt:

```
alert.accept()
```

Up to now, these methods could be used on an alert type of user prompt too. The difference between a ‘confirm’ and an ‘alert’ si that the ‘confirm’ user prompt also has a ‘Cancel’ button, which can be clicked from a test. Doing so is done by calling the ‘dimiss()’ method on the ‘alert’ variable:

```
alert.dismiss()
```

In case only one interaction with the ‘confirm’ is required in the test, there is no need to create a new variable only for the one interaction. Instead, the desired methods for interactions can be called as follows (this is the example for clicking ‘OK’):

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

Similarly, closing the ‘confirm’ by clicking the ‘Cancel’ button in a one-liner can be done:

```
driver.switchTo().alert().dismiss();
```

Example

Scenario 1.1. Click the button which triggers a ‘confirm’ to be displayed. Check that the text on the ‘confirm’ is ‘Confirm modal’. Click the ‘OK’ button on the ‘confirm’. Check that the field which displays what the user clicked now displays ‘OK’.

The first step is to click the button which triggers the ‘confirm’ to be displayed. Then, the handle to the newly opened ‘confirm’ will be stored to a variable of type ‘Alert’.

```
page.confirmButton.click();
Alert alert = driver.switchTo().alert();
```

Checking that the ‘confirm’ text is the expected one is done via the ‘getText()’ method:

```
assertEquals("Confirm modal", alert.getText());
```

Now, the ‘confirm’ can be closed, by clicking the ‘OK’ button.

```
alert.accept();
```

At this point, in this test, in the paragraph whose corresponding WebElement is named ‘userClicked’, the text ‘OK’ is displayed. To check this, the following assertion will be used:

```
assertEquals("OK", page.userClicked.getText());
```

Scenario 1.2. After scenario 1.1 is executed successfully, in the same test, open the ‘confirm’ again. Close the ‘confirm’ by clicking the ‘Cancel’ button. Check that the field which displays what the user clicked now displays ‘Cancel’.

Opening the ‘confirm’ again is done just as the first step of Scenario 1.1.:

```
page.confirmButton.click();
```

Close the ‘confirm’ by clicking the ‘Cancel’ button. This is done by using the ‘dismiss()’ method from Selenium. Now, think about what happened previously: first you opened the ‘confirm’ and you got a handle to it. Then you closed it. Now it opens again. Luckily there is no need to regenerate the handle to the ‘confirm’ (like it is the case with switching to windows). Once you created the ‘alert’ variable for the ‘confirm’ you can reuse that variable even if you closed, then reopened the ‘confirm’.

```
alert.dismiss();
```

Now, since the ‘Cancel’ button was clicked, check that the field which displays what the user clicked now displays ‘Cancel’:

```
assertEquals("Cancel", page.userClicked.getText());
```

Scenario 2. Open a ‘confirm’ type of user prompt. Close it by clicking the ‘Cancel’ button. Check that the paragraph which displays what the user clicked shows the text ‘Cancel’.

This is a short test, since there is only one interaction with the ‘confirm’ and there is no need to store the ‘confirm’ handle to a variable:

```
page.confirmButton.click();
driver.switchTo().alert().dismiss();
assertEquals("Cancel", page.userClicked.getText());
```

10.3. Prompt

The prompt type of user prompt:

- Displays:
  - A text
  - An input field, with or without a predefined text
  - An OK button
  - A Cancel button
- Has the following JS code: **prompt**("thePromptMessage"); or **prompt**("thePromptMessage", “thePromptInputField”);
- Can be interacted with from Selenium as:
  - Clicking the OK button (accept)
  - Clicking the Cancel button (dismiss)
  - Reading its text
  - Typing in the input field

In case more than one interaction with the ‘prompt’ is made in a test, it is worth storing a handle to the ‘prompt’ by creating an ‘Alert’ object variable. As mentioned in the previous subchapter, no matter what type of user prompt appears in the test, the type of variable used for storing its handle will be ‘Alert’. Storing the handle is done just as for the other types of user prompts:

```
Alert alert = driver.switchTo().alert();
```

Retrieving the text of the ‘prompt’ is also done just as for the ‘alert’ and ‘confirm’ from the previous subchapter:

```
alert.getText()
```

It is the same with clicking on the ‘OK’ button of the prompt:

```
alert.accept()
```

Clicking the ‘Cancel’ button of the ‘prompt’ is done by calling the ‘dimiss()’ method on the ‘alert’ variable:

```
alert.dismiss()
```

Aditionally, the user can type in an input field displayed on the ‘prompt’. This can be done using the ‘sendKeys()’ method from Selenium. Note that at the time of writing this book, this functionality does not work on ChromeDriver(). Its’ usage is:

```
alert.sendKeys(“textToType”)
```

Just as before, if there is no need to create a new variable in the test, the ‘prompt’ related methods can be called directly in the form:

```
driver.switchTo().alert().dismiss();
```

Example

Scenario 1. Open a new ‘prompt’ and check that the text displayed on it is ‘Type something’. Click the ‘OK’ button on the prompt, and check that the paragraph which displays what the user clicked now shows the text ‘OK’. Open the ‘prompt’ again, then close it by clicking the ‘Cancel’ button and check that the paragraph which dispays what the user clicked now shows ‘Cancel’.

Just as in the previous examples, for ‘alerts’ and ‘confirms’, first a button is clicked which triggers the ‘prompt’ to be displayed. Then, the handle to the ‘prompt’ is stored in an ‘Alert’ variable.

```
page.promptButton.click();
Alert alert = driver.switchTo().alert();
```

Checking the text of the ‘prompt’ and closing it (via the ‘Ok’ button) are covered by the following code:

```
assertEquals("Type something", alert.getText());
alert.accept();
```

Now the check that the paragraph displaying what the user clicked shows the text ‘OK’ is done through this assertion:

```
assertEquals("OK", page.userClicked.getText());
```

The next steps of this test are to reopen the prompt, then close it by clicking the ‘Cancel’ button, and finally checking that the label displaying what button was clicked is correct:

```
age.promptButton.click();
alert.dismiss();
assertEquals("Cancel", page.userClicked.getText());
```

Scenario 2. Open a new 'prompt' and close it by clicking the 'Cancel' button. Check that the paragraph which dispays what the user clicked now shows 'Cancel'.

The code which covers this scenario is similar to previous subchapters:

```
page.promptButton.click();
driver.switchTo().alert().dismiss();
assertEquals("Cancel", page.userClicked.getText());
```

GitHub location of example code

- PageObject class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/java/tutorialsolution/pages/UserPromptsPage.java
- Test class: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/test/java/tutorialsolution/userprompts/UserPromptsTest.java
- Test methods: alertGetTextAndAccept, alertAccept, confirmGetTextAcceptAndDismiss, confirmAcceptAndDismiss, promptGetTextAcceptAndDismiss, promptAcceptAndDismiss
- HTML code: https://github.com/iamalittletester/selenium-tutorial/blob/master/src/main/resources/userPrompts.html

Chapter references

W3.org. (2019). *HTML 5.2: 7. Web application APIs*. [online] Available at: https://www.w3.org/TR/html5/webappapis.html#user-prompts [Accessed 22 Mar. 2019].

W3schools.com. (2019). *Window alert() Method*. [online] Available at: https://www.w3schools.com/jsref/met_win_alert.asp [Accessed 22 Mar. 2019].

W3schools.com. (2019). *Window confirm() Method*. [online] Available at: https://www.w3schools.com/jsref/met_win_confirm.asp [Accessed 23 Mar. 2019].

W3schools.com. (2019). *Window prompt() Method*. [online] Available at: https://www.w3schools.com/jsref/met_win_prompt.asp [Accessed 23 Mar. 2019].

Seleniumhq.github.io. (2019). *WebDriver*. [online] Available at: https://seleniumhq.github.io/docs/wd.html#javascript_alerts_prompts_and_confirmations [Accessed 23 Mar. 2019].

11. WebDriverWait

See attached powerpoint.