Selenium Fundamentals

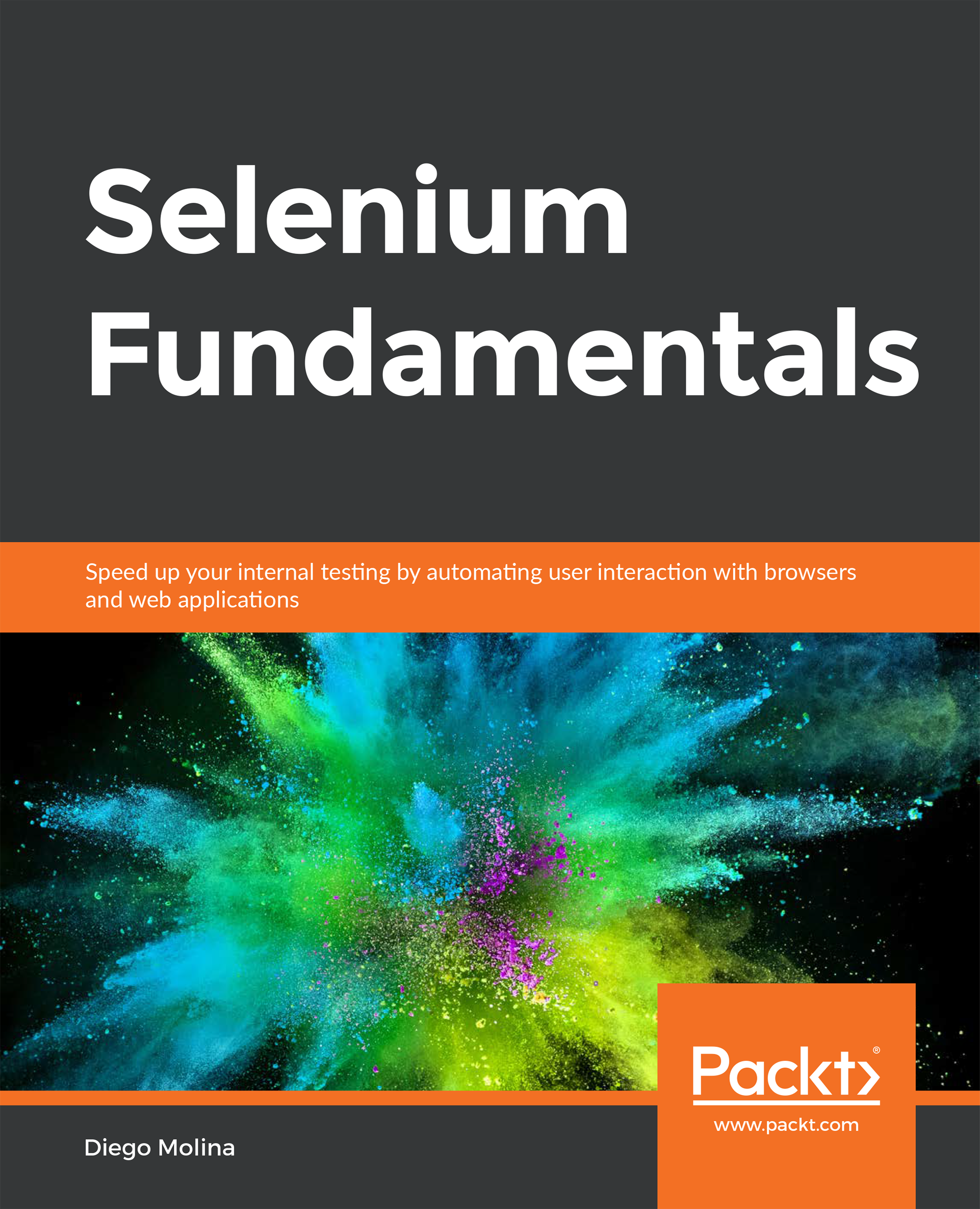
Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Acquisitions Editors:** Aditya Date  
**Content Development Editor:** Edwin Moses, Aritro Ghosh  
**Production Coordinator:** Samita Warang



# About the Author

**Diego Molina** is a software engineer in Test, specialized in testing tool development, advising, and training teams how to test better. He worked initially as a software developer for 8 years before switching completely to testing, where he has been during the last 7 years. He is one of the maintainers of the official Docker-Selenium project and a Selenium contributor.

He also co-created Zalenium, a dynamic, ready to use Selenium Grid. He spends most of his time working with different teams and finding ways to do UI testing in a more simple way. Additionally, he is often attending the Selenium conferences, either presenting or collaborating with the on-site workshops. You can find him often in the IRC/Slack channel for Selenium.

# Preface

Automated testing is more efficient than manual testing, and more and more industries in the IT sector are opting for automated tests for applications, whether desktop or web.

Selenium is the most popular tool used to automate browsers, and as it is currently on the path to become a W3C standard, its importance in the testing world will only grow. Selenium is open source, platform independent, and can be used to interact in a programmatic way with web applications through any major vendor browser (such as Google Chrome, Mozilla Firefox, Internet Explorer, Safari, Microsoft Edge, and Opera).

## Who This Book Is For

This book is designed for software quality assurance and development personnel who want to learn how to automate browser activity and web-based user interface tests with Selenium.

## What This Book Covers

[Chapter 1](https://my.safaribooksonline.com/9781789803815/585cbe81_ccad_4261_adbd_b6a025494f30_xhtml), Getting Started, deals with what automation is and why it is important. It then covers the how to use the Selenium WebDriver, and then finally focuses on how to configure the development environment for Selenium.

[Chapter 2](https://my.safaribooksonline.com/9781789803815/8353a458_db0d_4ee7_81ed_0699e99b0b42_xhtml), WebDriver Functionality, takes you deeper into the details of Selenium, covering each of the main WebDriver class methods and objects for the most commonly used browsers. Lastly, the chapter focuses on how to open and manipulate a browser window and navigate to a web page.

[Chapter 3](https://my.safaribooksonline.com/9781789803815/bd577ae9_55db_4f6a_81f2_2c44110dfe26_xhtml), WebElement Functionality, talks about how to use browser developer tools to review the code behind a web page and how to find and interact with elements to later write and perform tests on them.

[Chapter 4](https://my.safaribooksonline.com/9781789803815/108d1a2e_e4f5_4d7b_b057_2a37e35d34ac_xhtml), Advanced Element Function, covers how to navigate, search, and identify different elements in a web application by using effective element locators.

[*Chapter 5*](https://my.safaribooksonline.com/9781789803815/c4d3f4b9_00f8_4655_8432_c29f6bbab144_xhtml), Waiting For Elements, trains the students on how to write a stable automation script by waiting for an element to be present. It discusses the differences between implicit and explicit waits, and finally shows how to effectively synchronize a script in a custom condition.

[Chapter 6](https://my.safaribooksonline.com/9781789803815/840c10a0_ba94_44a5_9796_2241a87fe1ee_xhtml), Page Object Model, covers the principles behind Page Object Model, and how to model and implement an automation script.

[*Chapter 7*](https://my.safaribooksonline.com/9781789803815/0d0bd469_4acd_4656_971f_d18e5eb771d2_xhtml), Writing Tests, talks about test frameworks and how to choose them; as well as how to create test scripts and how to validate results.

[Chapter 8](https://my.safaribooksonline.com/9781789803815/326642a1_a281_47cf_8efb_0967cb0212ab_xhtml), Analysis and Troubleshooting, discusses the root cause of a test failure by analyzing a test report, and tracking down errors It also covers how to separate real issues from flaky tests.

[Chapter 9](https://my.safaribooksonline.com/9781789803815/14e31acf_bb2c_4e51_882e_a3f287f6ac3f_xhtml), Using a Selenium Grid, focuses on how to configure and connect to the Selenium Local Grid, as well as the Selenium Network Grid. Afterwards, covering how to connect to third party services.

## To Get the Most out of This Book

To complete this book successfully, students will require computer systems with at least an Intel Core i5 processor or equivalent, 8 GB RAM, and 4 GB available storage space.  Along with this, you would require the following software:

* Operating System: Windows or above
* Browser: Google Chrome (latest version)
* Selenium WebDriver
* Google ChromeDriver
* IntelliJ IDEA

### Download the Example Code Files

You can download the example code files for this book from your account at [www.packt.com](http://www.packt.com). If you purchased this book elsewhere, you can visit [www.packt.com/support](http://www.packt.com/support) and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at [www.packt.com](http://www.packt.com).
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

* WinRAR/7-Zip for Windows
* Zipeg/iZip/UnRarX for Mac
* 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/TrainingByPackt/Beginning-Selenium/tree/master/docs>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

### Conventions Used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The WebDriver class provides constructors for each browser."

A block of code is set as follows:

import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.chrome.ChromeDriver;

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

private static void checkFrames(){  
  
 WebDriver driver = new ChromeDriver();  
  
 try {  
 driver.get("https://trainingbypackt.github.io/  
Beginning-Selenium/lesson\_2/exercise02\_concept\_04.html");  
  
 **driver.switchTo().frame("info");**

**Bold**: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "If a user clicks on the Your Account option on the Home Webpage, the application will check whether they have already logged in."

*Warnings or important notes appear like this.*

*Tips and tricks appear like this.*

## Get in Touch

Feedback from our readers is always welcome.

**General feedback**: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

**Errata**: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packt.com/submit-errata](http://www.packt.com/submit-errata), selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy**: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

**If you are interested in becoming an author**: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com/).

### Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit [packt.com](http://www.packt.com/).

# Getting Started

The more robust and complex web applications get, the more robust and complex the process of testing them becomes. As a crucial part of software development, the testing phase must be as bulletproof as possible, while also providing processes that simplify the management and execution of tests. In this chapter, you will learn what software automation is and how important it is to the development cycle. You will be introduced to Selenium WebDriver, the most commonly used browser automation tool, which is on its way to becoming a W3C standard. You will also learn how to configure a testing environment for Selenium WebDriver.

By the end of this chapter, you will be able to:

* Explain what automation is and why it is important
* Use the Selenium WebDriver
* Configure the development environment for Selenium

## Software Automation

Automation turns a manual process or system into a system that operates automatically. Once we have analyzed and dissected a process or system and can understand how it works (the input it requires, the way it processes that input, and its expected results), it is possible to design means that allow the process or system to operate automatically. First, one must decide whether it makes sense to automate something; this usually involves an analysis of the effort required to automate the process, how often the automation will be used, and the time consumed in maintaining the automation.

Here are some of the benefits of automation:

* It reduces time, allowing you to execute tasks more quickly.
* It reduces expenses; fewer resources are required to complete tasks.
* It increases the value of the company or team; collaborators can be assigned  
  to higher value tasks.
* Automated and scripted processes are less prone to errors.
* Automation is repeatable and reusable. It can be improved over time, little by little, to adapt to changes in the process or web application.

Automation is used in many industries, including the field of software design. Deploying and configuring an application, generating data for testing, and the testing itself are only some of the tasks that can be automated in a software development  
cycle.

### Going the Automation Way

Tedious, repetitive, and manual tasks are the first candidates for automation in a process. However, the automation itself requires time, money, resources, and personnel. So, it is key to understand whether the **Return on Investment** (**ROI**) of implementing the automation will be more beneficial than continuing to perform the tasks manually. Think of the most basic test involved in an application. Assume that it requires two teammates and six hours of time from each of them. If, by automating the test, we can reduce its execution time to only one hour, with only one collaborator in charge, then automation should be considered a priority (one added value would be the ability to reuse the test automation for testing similar applications in the future).

Continuing with the preceding example, what if implementing automation for the test will take a month, and will require a team of five testers? What if it is unlikely that we will be able to reuse the automation project in future applications? What if the test is so complex that the resources required to maintain the automation script will be considerably higher and more expensive? Depending on the answers to these questions, automation may not be the best option. There will be cases where analyzing the situation will not be as straightforward as in the cases mentioned previously.

The general process of deciding whether to automate a system should involve considering the answers to the following questions:

* How many resources and how much time are required to build the automation solution?
* How many resources and how much time are required to maintain the automation solution?
* How likely is it that the solution will be used in the middle and long term?
* What is the risk exposure level if the system isn't automated? (Even if the implementation of automation is extremely expensive in terms of resources and time, it might help to mitigate huge risks for the web application, in which case, it would make sense to implement it.)

### Activity: Automation Brainstorming

**Scenario**

Suppose a retailer approaches your company to build a shopping cart application for him. He's more interested to know the tasks that would be automated. You're to meet with him to brief him about it.

**Aim**

To list potential candidates for automation of tasks and processes involved in the software development cycle.

**Steps for Completion**

1. Write down at least five manual, tedious, or repetitive tasks, for which automation would be a good choice.  
   **Hint**: These could be along the lines of the payment process.
2. Analyze if automation is feasible for the tasks you noted. Ask the questions you saw in the *Going the Automation Way* subsection.

You should now understand what automation is and how it can benefit software development.

*To refer to the detailed steps, go to the Solutions section at the end of this book on Page no. 161.*

## Selenium Overview

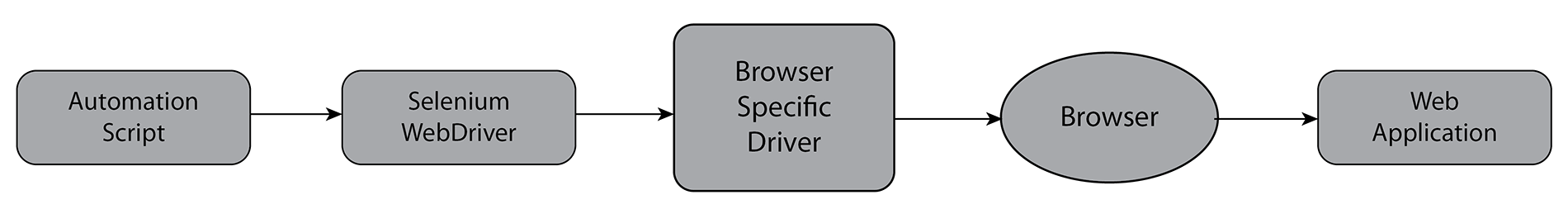
Selenium is one of the most commonly used frameworks for browser automation. As stated previously, tools like Selenium provide an easy and scalable way to test web applications and reduce time-consuming tasks.

Among other benefits, Selenium provides testing teams with the following:

* A standardized API that allows for automating web browsers
* A server application that helps to coordinate and distribute tests across multiple operating systems and browser combinations

In Selenium, test automation works through the following means:

1. Automation scripts are written by the testing and development teams. These scripts contain instructions for how to perform the test, including finding and manipulating objects on the web page, using information as input, verifying messages from the web application, and interacting with elements.
2. The Selenium WebDriver provides interfaces, classes, and methods to interact with browsers and elements in the web application.
3. Browser providers and third parties have developed browser drivers to work with the WebDriver API. This has made Selenium available to execute tests on all of the popular web browsers.



### The WebDriver API

To simplify its usage, the WebDriver API has official bindings in several different programming languages, including Java, Ruby, Python, C#, and JavaScript. Additionally, there are several third-party bindings in more languages, such as PHP and Golang. This means that the WebDriver API can be used in almost every software development environment. The WebDriver API is normally used to implement UI regression tests, and any other tasks that can be performed via a web browser. As stated previously, the WebDriver API allows for working with many browsers and many different operating systems, which covers most of the development and real-life scenarios of web applications.

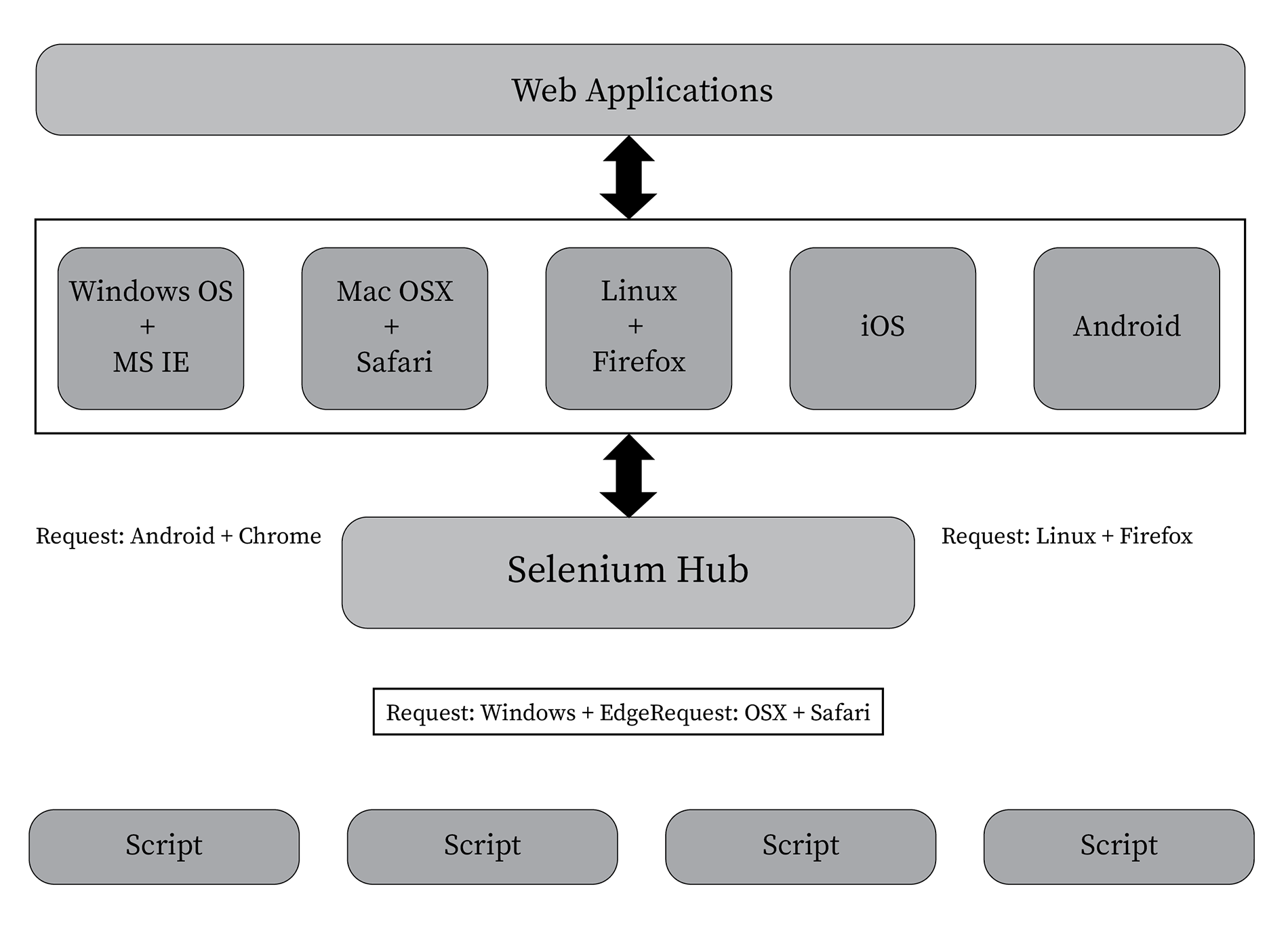
*More information on the W3C WebDriver specifications can be found at*[*https://www.w3.org/TR/webdriver/*](https://www.w3.org/TR/webdriver/)*.*

### The Selenium Server

After installing the Selenium Server, it is possible to test local and remote web applications by using any of the browsers installed in the Server. For small teams, a single server might be enough to test all of the team's applications. However, for teams that have to deal with a considerable amount of web applications, Selenium provides an option called Selenium Grid. As of Selenium 2.0, the Server now incorporates the functionality of Selenium Grid.

As you can see in the following diagram, test automation scripts can request to be run on a specific combination of operating systems and browsers. The Selenium Hub distributes those requests to the suitable nodes. Nodes are (physical or virtual) instances of the Selenium Server that allow you to test applications under a wide variety of operating systems (Windows, macOS, Android, and so on) and browsers (Internet Explorer, Chrome, Safari, and so on).

A grid consists of several systems running the server software in a hub and node configuration. We will take a more thorough look at the Selenium Grid in [Chapter 9](https://my.safaribooksonline.com/9781789803815/14e31acf_bb2c_4e51_882e_a3f287f6ac3f_xhtml), *Using a Selenium Grid*.



#### Selenium Glossary

The Selenium Glossary includes the following:

* **Selenium IDE**: This is a browser plugin that records user activity inside the browser with Selenium commands, which can be used to create test cases. The test cases can be played back afterwards, as a part of the test suite. As a starting point, it is useful to create test cases and to learn the Selenium syntax.

*For more information on the Selenium IDE, you can visit*[*https://seleniumhq.github.io/docs/quick.html#selenium\_ide*](https://seleniumhq.github.io/docs/quick.html#selenium_ide)*.*

* **Selenium WebDriver**: This is an API that provides a simple interface to interact with browsers in a programmatic way. This book mainly focuses on WebDriver; more details are provided at <https://www.w3.org/TR/webdriver/>.
* **Selenium Grid**: This enables your team to run tests in parallel, across different machines (physical or virtual). On each machine, there can be various browsers (in different versions) and different operating systems. More information can be found in the last chapter of this book, as well as at <https://seleniumhq.github.io/docs/quick.html#selenium_grid>.

*Selenium IDE is a Firefox add-on that allows for recording and playing back the execution of a script. The original IDE is no longer maintained, and, from Firefox 55, it does not work anymore. A new IDE (IDE2) is currently under development, in a "Work in Progress" status. Nevertheless, Selenium IDE is not recommended for middle-and long-term, maintainable testing code.*

### Activity: Selenium Planning

**Scenario**

Suppose that the shopping cart web application has been developed and is now in the hands of the testing team. Prior to writing test cases and scenarios, it is necessary to plan how Selenium will be used.

**Aim**

To identify business requirements and plan well how to go about using Selenium for a particular web application.

**Steps for Completion**

1. Identify some business requirements where test cases can be automated.
2. List the applications or services required for WebDriver to test the web-application:  
   **Hint**: This might include the version of the programming language that you intend to use to write the automation scripts, the browsers' drivers, the specific Selenium libraries, and so on.

*To refer to the detailed steps, go to the Solutions section at the end of this book on Page no. 162.*

## Environment Configuration

Before using the browser drivers in Selenium, the following prerequisites must be installed on the computer being set as the testing environment:

* Java 8 SDK
* Any compatible IDE (we recommend IntelliJ IDEA)

### Browser Drivers

As stated previously, Selenium WebDriver provides a standardized API that allows us to automate browser activity. Each browser provider and third-party project has developed its own specific browser driver, in order to be compatible with Selenium WebDriver.

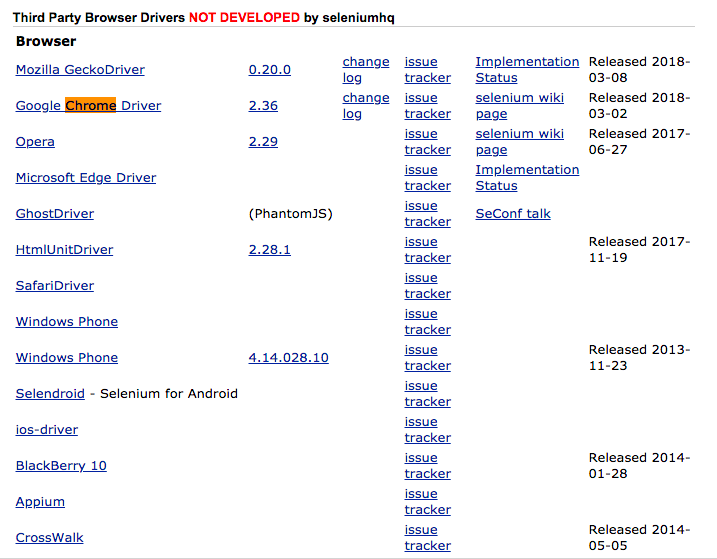
For example, if a testing team plans to automate tests with Selenium WebDriver and execute the scripts in Chrome, they will have to download its specific driver application in advance. This must be done for all of the browsers that the team wants  
to use.

GeckoDriver (Firefox), ChromeDriver (Chrome), and IE Driver (Internet Explorer) are the most frequently used drivers, and they follow the guidelines of the Selenium framework.

### Downloading and preparing browser drivers

Here, we shall download and install the drivers for the Chrome browser. The aim is to understand how to download and install browser drivers as a prerequisite of Selenium automation. The steps for completion of this process are as follows:

1. Browse to <http://www.seleniumhq.org/download/> to download the browser drivers from SeleniumHQ.
2. Download Google Chrome Driver:



*Downloading the IE and Firefox drives is optional, as we will only use Chrome throughout the remainder of the book.*

1. Unzip the downloaded files into a folder of your preference (such as C:\BrowserDrivers).
2. Optionally, add the folder referenced in the previous step (for example, C:\BrowserDrivers) to the system path. This will allow the driver to start without the need to include its location in every automation script that uses it.
3. For Windows, run the following command on a console window:

set PATH=$PATH;\path\to\chromedriver

  For Linux and macOS, run the following command on a console window:

export PATH=$PATH:/path/to/chromedriver

## Maven Project

Regardless of the chosen programming language, it is necessary to make sure that our IDE can import and use the Selenium libraries before writing Selenium scripts. With Java, the simplest way to do this is to use Maven. Maven is a project/dependency management tool. Most Java-compatible IDEs (such as IntelliJ IDEA) include support for using Maven.

With Maven, we will create a project that downloads and uses the required Selenium libraries and all of Selenium's dependencies, and then creates a POM.xml file. We will later import that file from our preferred IDE while creating our development project. For the purposes of this book, we will focus on the dependencies section of the POM.xmlfile. Here, we will specify the external libraries required by the project.

### Creating a Maven project

Here, you'll create a Maven project that can be used to write and perform test automation, using the Selenium WebDriver. Before you begin, verify that Java 8 SDK is installed on your computer. Then, use your preferred IDE (we recommend IntelliJ IDEA). Finally, download Maven from <https://maven.apache.org/download.cgi>. The steps for completion of this process are as follows:

1. Open the IDE.
2. Create a new Maven project by navigating to **File** | **New** | **Project**.
3. Select **Maven**, and click on **Next**.
4. Enter com.beginningselenium as GroupId and lesson1 as the ArtifactId, and click on Next. Finally, click on Finish.  
   Next, we'll look at adding Selenium WebDriver to the Maven project's POM.xml file.
5. The POM.xml file should open automatically. Before the </project> tag, add the following, and then save the file:

<dependencies>  
 <dependency>  
 <groupId>org.seleniumhq.selenium</groupId>  
 <artifactId>selenium-java</artifactId>  
 <version>3.11.0</version>  
 </dependency>  
</dependencies>

1. Right-click on the project, then navigate to Maven | Reimport.
2. Right-click on the src/main/java folder and navigate to New | Package.
3. Enter com.beginningselenium.selenium as the name, and click OK.
4. Right-click on the new package and navigate to New | Java Class.
5. Enter BrowserCheck for the name, and click on OK.

Configuring the development environment for Selenium consists of three simple steps, as follows:

* Installing and configuring the Java SDK
* Installing and configuring an IDE
* Downloading and configuring the browser drivers for all of the required browsers
* Using Maven, or any other means, to ensure that the programming language and IDE will be able to import and use the required Selenium libraries and documentation

### Verifying That the Development Environment Is Ready

Now that you have learned how to prepare browser drivers, it's time to verify that the development environment is ready to be used. The aim is to create an application that will verify that the development environment has been configured correctly. Before you begin, first ensure that you have successfully followed the steps in the sub-sections from previous sections. Also check whether the Chrome browser is installed on your computer.

The steps for completion of this process are as follows:

1. Import WebDriver and ChromeDriver to BrowserCheck.java:

import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.chrome.ChromeDriver;

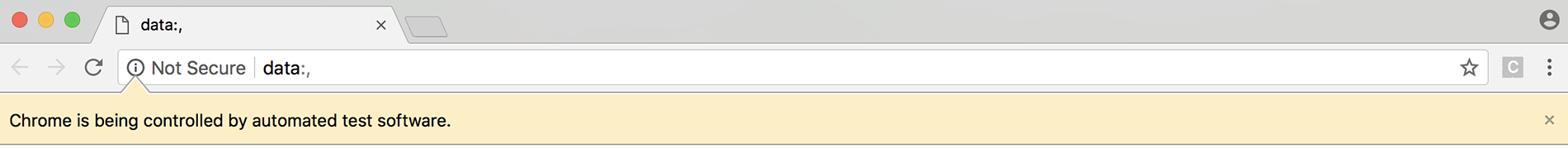
1. In the BrowserCheck class, enter the code to open and close the Chrome browser:

public static void main(String[] args) {  
 // Open and close the Chrome Browser  
 System.setProperty("webdriver.chrome.driver", "c:/  
browserdrivers/chromedriver.exe");  
 WebDriver chrome = new ChromeDriver();  
 chrome.quit();  
}

1. Run the application by navigating to Run | Run… | BrowserCheck.

If everything is configured correctly, the application will open and close Chrome.

This is what your browser should look like:



## Summary

You should now have a better understanding of what automation is and how it can be used in testing web applications. We have looked at the different components of a Selenium project. We have also configured a browser driver, created a Maven project that implements Selenium WebDriver, and created a Java application that opens and closes a browser window.

In the next chapter, we'll dig deeper into Selenium concepts by working with the main WebDriver class methods.

# WebDriver Functionality

In [*Chapter 1*](https://my.safaribooksonline.com/9781789803815/585cbe81_ccad_4261_adbd_b6a025494f30_xhtml)*, Getting Started*, we provided an overview of automation and its benefits, and we also covered Selenium and its components.

In this chapter, we will go deeper into the details of Selenium, describing each of the main WebDriver class methods. Upon completion of this chapter, students will be able to instantiate WebDriver objects for the most commonly used browsers, as well as open and manipulate a browser window and navigate to a web page.

By the end of this chapter, you will be able to:

* Instantiate a WebDriver object to use Selenium
* Work with the most frequently used methods of the WebDriver class
* Navigate, resize, and switch between windows

## Instantiating a WebDriver Chrome

There are many available (Chrome, Safari, IE, Edge, Firefox, Opera, and so on) to navigate the internet, and each of them is built with its own features, specific approaches, and technologies. This means that a web application might behave slightly different on Google Chrome than it does on Internet Explorer. With this in mind, the WebDriver class can be instantiated according to the browser that we want to perform the test on.

In *Environment Configuration*, from the previous chapter, you learned how to prepare browser drivers. Once you've prepared the browser drivers, you can use any browser with any given automation script. The WebDriver class provides constructors for each browser. As we will only use Chrome throughout this book, we will focus on ChromeDriver. We can instantiate a WebDriver variable for use on Chrome as follows:

WebDriver driver = new ChromeDriver();

*If you did not set the ChromeDriver on a system path (covered in the third section of this chapter), you will also have to let ChromeDriver know the location of your Chrome for every automation script that you write.*

If you have set your ChromeDriver to the system path, you can skip the following step:

ChromeOptions options = new ChromeOptions();  
options.setBinary("/path/to/location");  
WebDriver driver = new ChromeDriver(options);

You can also use ChromeOptions to start Chrome with a Chrome extension, as follows:

ChromeOptions options = new ChromeOptions();  
options.addExtension("example.crx")

### Creating the Main Structure of Your First Selenium Automation Script

Here, we shall be creating the main structure of the script, import the required libraries, and instantiate our WebDriver object. Before you begin, make sure that you have followed the steps in *Environment Configuration*, of the previous chapter. Then, use IntelliJ IDEA for the creation of a Selenium automation script. The steps for completion of this process are as follows:

1. Import, via WebDriver, WebElement, and ChromeDriver, the minimum libraries required for a Selenium script to run:

import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.chrome.ChromeDriver;

1. Create the main class of the script, as follows:

public class ActivityLesson02 {  
}

1. Inside of the ActivityLesson02 class, create a method that will instantiate the driver. Add the quit call, in order to end the session, and then close the browser:

public static void main(String[] args) {  
 activityLesson02AutomationScript();  
}  
  
private static void activityLesson02AutomationScript(){  
  
 WebDriver driver = new ChromeDriver();  
  
 // the other script code will go here  
 driver.quit();  
}

1. Compile and execute the script. As in the previous chapter, you should see the Chrome browser open and close.
2. Save the file as Activity01Lesson02.java.

You should now know how to instantiate a WebDriver variable for Chrome.

## An Overview of Frequently Used Methods

As we mentioned in the first chapter, Selenium WebDriver is an API that allows for automating browser activity.

*Updated information about the class can always be found at*[*https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/WebDriver.html*](https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/WebDriver.html)*.*

The following list provides descriptions of the most frequently used methods of the WebDriver class:

* void close(): This closes the current window and exits the browser (if it's the last window currently open).
* WebElement findElement(By by): This receives search parameters and returns the first matching element on the current page, or NoSuchElementException if no element is found. We will cover this subject in detail in upcoming chapters.
* java.util.List<WebElement> findElements(By by): This receives search parameters and returns all of the matching elements on the current page, or an empty list if no element is found. We will cover this subject in detail in upcoming chapters.
* void get(java.lang.String url): This loads the received URL in the current browser window. It is used to indicate and load the web application that we want to test.
* java.lang.String getCurrentUrl(): This returns the URL of the current browser.
* java.lang.String getPageSource(): This returns the source of the last loaded page. Depending on the driver being used, if the page is modified after having been loaded, the returned source might not correspond to the modified page.
* java.lang.String getTitle(): This returns the title of the current page, or null if one has not been set. It is normally used to verify that we are working in the correct window during a test.
* WebDriver.Options manage(): This returns the interface option of the menu. For example, it allows for the management of cookies (add, delete, and get).
* void quit(): This quits the driver and closes all associated windows. It is the preferred method to end an automation script.

You should now have a better understanding of the most frequently used methods of the WebDriver class.

### Activity: Starting and Finalizing a Script

**Before You Begin**

* Make sure that you have followed the steps in *Creating the Main Structure of Your First Selenium Automation Script*.
* Continue working on the Activity01Lesson02.java from *Creating the Main Structure of Your First Selenium Automation Script*.

**Scenario**

In the previous section, we created the main structure of the script, imported the required libraries, and instantiated our WebDriver. Now, we want to use some of the WebDriver methods to start and finalize the script.

**Aim**

To use the driver object to navigate to a desired web application URL.

**Steps for Completion**

1. Open Activity01Lesson02.java that you used in the *Creating the Main Structure of Your First Selenium Automation Script*.
2. Use the get() method with the driver object to navigate to the desired web application URL. We'll use <https://www.google.com> in this activity.
3. Compile and execute the script.
4. Navigate to <http://www.google.com>. When the page has finished loading, the browser will close.

*To refer to the detailed steps, go to the Solutions section at the end of this book on Page no. 162.*

## Controlling the Browser Window

A web application starts as a single page in a browser window, but new windows and popups can be opened and closed throughout the workflow. The WebDriver class provides methods to handle navigating, resizing, and switching between windows.

### Navigation

The navigation commands are related to what a user does when he/she interacts with the browser and goes to a previous page, refreshes a page, or browses to a given URL. The navigation interface provides the ability to perform these actions.

The WebDriver class provides the WebDriver.Navigation navigate() method to navigate to URLs during the execution of a script. It returns the browser's history and allows you to navigate to any given URL. The interface includes methods to navigate. The following are some examples of how this method is used:

* navigate().back(): This goes to the previous page in the browser history
* navigate().forward(): This moves forward to the next page in the browser history
* navigate().refresh(): This reloads the current page
* navigate().to("<http://www.yahoo.com>"): This browses to an indicated URL

#### Navigating in an Automation Script

The aim here is to navigate between URLs in an automation script. You'll be creating an automation script that navigates between websites. Before you begin, first make sure that you have followed the steps in the previous activity of this chapter. Then, continue to work on the same file from the same activity of this chapter. Finally, include the command that follows in the ActivityLesson02 constructor.

This will add a wait of five seconds for the pages to load, before attempting to verify the lines of the test. We will review this subject later, in [*Chapter 5*](https://my.safaribooksonline.com/9781789803815/c4d3f4b9_00f8_4655_8432_c29f6bbab144_xhtml)*, Waiting for Elements*:

driver.manage().timeouts().implicitlyWait(5, TimeUnit.  
SECONDS);

The steps for completion of this process are as follows:

1. Using driver.get();, load any known website:

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

1. Use getTitle() to verify that the title of the window is equal to the one from the website that you just navigated to (a simple comparison, with an if statement, should be sufficient):

driver.getTitle().equalsIgnoreCase("Google")

1. Using the System.out.println method, display messages that indicate whether the verification was successful:

{  
 System.out.println("Script worked, the  
title contains 'Google'");  
 } else {  
 System.out.println("Something went wrong  
with the script, 'Google' was not found");  
 }

1. Using navigate().to(""), browse to a new website:

driver.navigate().to("http://www.yahoo.com");

1. Use driver.getTitle() to verify that the title in the window is the same as the one from the website that you just navigated to (a simple comparison, with an if statement, should be sufficient):

if (driver.getTitle().equalsIgnoreCase("Google"))

1. Using the System.out.println method, display messages that indicate whether the verification was successful:

{  
 System.out.println("Script worked, the  
title contains 'Yahoo'");  
 } else {  
 System.out.println("Something went wrong  
with the script, 'Yahoo' was not found");  
}

1. Using navigate().back(), navigate back to Google:

driver.navigate().back();

1. Use driver.getTitle() to verify that the title in the window is the same as the one from the website that you just navigated to (a simple comparison, with an if statement, should be sufficient):

if (driver.getTitle().equalsIgnoreCase("Google"))

1. Using the System.out.println method, display messages that indicate whether the verification was successful:

{  
 System.out.println("Script worked, the  
title contains 'Google'");  
 } else {  
 System.out.println("Something went wrong  
with the script, 'Google' was not found");  
}

1. Compile and run the script.

### Resizing

When we open a browser through Selenium, it will start with the default settings. Depending on our objective, it might be necessary to change the browser window size. For example, if we have a responsive web application, we might want to change the browser size to automatically check how the application behaves in different window sizes.

The WebDriver.Options manage() method is provided by the WebDriver class to resize windows during the execution of a script. The manage().window() interface includes methods that allow us to resize the windows.

Here are some of the methods provided by the manage().window() interface:

* manage().window().maximize(): Maximizes the current window
* manage().window().fullscreen(): Sets the current window to full screen size
* manage().window().setPosition(new Point(50, 200)): Sets the position of the current window
* manage().window().setSize(new Dimension(300, 500)): Sets the size of the current window
* manage().window().getPosition(): Gets the position of the current window
* manage().window().getSize(): Gets the size of the current window

#### Resizing Windows in an Automation Script

Here, we'll be creating an automation script that resizes windows. The steps for completion of this process are as follows:

1. Using the get() method in the ActivityLesson02 constructor, load any known website:

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

1. Using the manage().window().setSize(Dimension) method, resize the current window:

driver.manage().window().setSize(new Dimension(300,500));

1. Using the manage().window().getSize() method, verify that the size of the window is equal to the size of the one that you set in *step 2*. The manage().window().getSize(Dimension) method also provides the getSize().getHeight() and getSize().getWidth() methods:

if (driver.manage().window().getSize().getHeight() == 300  
&& driver.manage().window().getSize().getWidth() == 500 )

1. Using the System.out.println method, display messages that indicate whether the verification was successful:

{  
 System.out.println("Load script worked, the  
window was resized");  
 }  
 else {  
 System.out.println("Something went wrong  
with the script, the window was not resized to the desired  
size");  
 }

1. Using the manage().window().maximize() method, maximize the current window:

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

1. Using the manage().window().getSize() method, verify that the size of the window is different from the one that you set in *step 2*. The manage().window().getSize method also provides the getSize().getHeight() and getSize().getWidth() methods:

if (driver.manage().window().getSize().getHeight() != 300  
&& driver.manage().window().getSize().getWidth() != 500 )

1. Using the System.out.println method, display messages that indicate whether the verification was successful:

{  
 System.out.println("Load script worked, the  
window was resized");  
 }  
 else {  
 System.out.println("Something went wrong  
with the script, the window was not resized");  
 }

1. Compile and run the script.

### Managing Alerts

Web applications can include pop-up alerts (modal dialogs) during any part of the process. These alerts can be of the following three types:

* A simple alert that will show a message and expect the user to click on a button to continue an ongoing process
* A confirmation alert that will show a message and expect the user to accept or dismiss the alert before continuing with an ongoing process
* A prompt alert that will show a message and expect the user to put a value in before continuing with an ongoing process

The WebDriver class, combined with the WebDriver.TargetLocator and Alert interfaces, allows automation scripts to handle the different these types of alerts. The WebDriver.TargetLocator switchTo() method is used to select a frame or a  
window.

The WebDriver.TargetLocator interface provides the following methods to manage frames or windows:

* activeElement(): Switches to the element that currently has the focus
* alert(): Switches to the active modal dialog
* defaultContent(): Selects either the first frame on the page, or the main document when a page contains iframes
* frame(int index): Selects a frame by its (zero-based) index
* frame(String nameOrID): Selects a frame by its name or ID
* frame(WebElement frameElement): Selects a frame if its corresponding WebElement was previously located
* parentFrame(): Changes the focus to the parent context
* window(String nameOrHandle): Switches the focus to a window, by its name or handle

The Alert interface provides the following methods to manage modal dialogs:

* accept(): Equivalent to clicking on the Accept button on a modal dialog
* dismiss(): Equivalent to clicking on the Cancel button on a modal dialog
* getText(): Gets the text of a modal dialog
* sendKeys(String keysToSend): Sends specific keys to the modal dialog (used when the modal dialog requires the user to input data)

#### Managing Alerts in an Automation Script

Here, we'll be creating an automation script that manages alerts. We'll create three different test classes, to test different modal dialogs.

The steps for completion of this process are as follows:

1. Review and analyze the structure of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_2/exercise02_concept_03.html> file.
2. Open the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_2/exercise02_concept_03.html> page; navigate around the page and make sure that you understand it, so that you can analyze its behavior.
3. Create a new Java file for the automation script using IntelliJ IDEA. Make sure that you include the required libraries for the script to work.
4. For the simple alert modal dialog, create a method and name it alertsChecks:

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

1. Inside the alertsChecks method, create a variable of type Alert, and use the driver.switchTo().alert method to assign it to the modal dialog (as highlighted in the following code snippet):

private static void alertsChecks(){  
 WebDriver driver = new ChromeDriver();  
 try {  
  
 driver.get("https://trainingbypackt.github.io/  
Beginning-Selenium/lesson\_2/exercise02\_concept\_03.html");  
  
 // Accepting the first alert  
 **Alert simpleAlert = driver.switchTo().alert();**  
System.out.println("Alert text contents: " + simpleAlert.  
getText());

1. Verify that the message on the modal dialog is: "Welcome! This is a simple alert. Press 'Accept' to continue":

if (simpleAlert.getText().equalsIgnoreCase("Welcome! This  
is a simple alert. Press 'Accept' to continue"))

1. Using the System.out.println method, display a message that indicates whether the verification was successful:

{  
 System.out.println("It worked, the expected  
simple alert was shown");  
 }  
 else {  
 System.out.println("Something went wrong,  
the expected simple alert was NOT shown");  
 }

1. Accept the alert:

simpleAlert.accept();

1. Now, create a variable of type Alert, and use the driver.switchTo().alert method to assign it to the modal dialog:

Alert confirmAlert = driver.switchTo().alert();

1. Either accept or dismiss the alert:

if (confirmAlert.getText().equalsIgnoreCase("This is a  
confirm alert. Do you want to accept or cancel?"))

1. Using the System.out.println method, display a message that indicates whether the alert was accepted or dismissed.

{  
 System.out.println("It worked, the expected  
confirmation alert was shown");  
 }  
 else {  
 System.out.println("Something went wrong,  
the expected confirmation alert was NOT shown");  
 }

1. Accept the alert:

confirmAlert.accept();

1. Now, create a variable of type Alert, and use the driver.switchTo().alert() method to assign it to the modal dialog:

Alert promptAlert = driver.switchTo().alert();

1. Populate the input field with your desired answer:

promptAlert.sendKeys("Java");

1. Accept the alert:

promptAlert.accept();

1. Verify that the title of the page includes the text while populating the input field:

if (driver.getTitle().contains("Java"))

1. Using the System.out.println method, display a message that indicates whether the text was included:

System.out.println("It worked, the expected prompt alert  
was shown and the text was included in the title.");  
 } else {  
 System.out.println("Something went wrong,  
the expected prompt alert did not work");  
 }

1. Compile and execute the script.

### Managing Frames and iFrames

Web applications can include frames that divide pages into different sections. Each of the sections usually includes specific content. The **WebDriver** class, combined with the **WebDriver.TargetLocator** interface, allows automation scripts to manage and iframes.

*See the Managing Alerts section for a detailed description of the switchTo() method and WebDriver.TargetLocatorinterface.*

#### Managing Frames in an Automation Script

Here, we'll be creating an automation script that manages frames and iframes. The steps for completion of this process are as follows:

1. Review and analyze the structure of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_2/exercise02_concept_04.html> file in the Chrome browser.
2. Create a new Java file for the automation script. Make sure you include the required libraries for the script to work.
3. For the frames, create a method and name it checkFrames:

checkFrames();

1. Using the driver.switchTo().frame() method, change the focus to the frame with the ID info (the focus can also be changed to a certain frame with its name, if it has one):

private static void checkFrames(){  
  
 WebDriver driver = new ChromeDriver();  
  
 try {  
 driver.get("https://trainingbypackt.github.io/  
Beginning-Selenium/lesson\_2/exercise02\_concept\_04.html");  
  
 **driver.switchTo().frame("info");**

If a frame does not have an ID or a name, it can be selected by its (zerobased) index.

1. With the focus on the info frame, use the driver.getPageSource() method to obtain the frame's HTML contents, and verify that it contains the string "Frame Info":

if (driver.getPageSource().contains("Frame Info"))

1. Using the System.out.println method, display a message that indicates whether the focus is on the info frame.

{  
 System.out.println("The script worked, the   
focus was changed to Frame Info");  
 } else {  
 System.out.println("Something went wrong   
with the script, the focus was not changed to Frame Info");  
 }

1. Using the driver.switchTo().defaultContent() method, return the focus to the main page:

driver.switchTo().defaultContent();

1. Using the driver.switchTo().frame() method, change the focus to the frame with the ID "title".

driver.switchTo().frame("title");

1. With the focus on the title frame, use the driver.getPageSource() method to verify that the HTML content of the frame contains Frame Title:

if (driver.getPageSource().contains("Frame Title"))

1. Using the System.out.println method, display a message that indicates whether the focus is on the title frame.

{  
 System.out.println("The script worked, the  
focus was changed to Frame Title");  
 }   
 else {  
 System.out.println("Something went wrong  
with the script, the focus was not changed to Frame Title");  
 }

1. Using the driver.switchTo().defaultContent() method, return the focus to the main page.
2. For the iframes, create a method and name it checkIFrames:

checkIFrames();

1. Using the driver.switchTo().frame() method, change the focus to the frame with the ID twitter (the focus can also be changed to a frame with its name, if it has one):

driver.switchTo().frame("twitter");

When a frame does not have an ID or a name, it can be selected by its (zerobased) index.

1. With the focus on the twitter frame, use the driver.getPageSource() method to verify that the content of the page includes "Frame Twitter":

if (driver.getPageSource().contains("Frame Twitter"))

1. Using the System.out.println method, display a message that indicates whether the focus is on the twitter frame.

{  
 System.out.println("The script worked, the  
focus was changed to iFrame Twitter");  
 } else {  
 System.out.println("Something went wrong  
with the script, the focus was not changed to m");  
 }

1. To change the focus from the twitter frame to the <iframe> embedded within it, we will have to locate the web element with the findElement method (we will explain this subject in an upcoming chapter):

WebElement twitterFrame = driver.findElement(By.  
tagName("iframe"));  
driver.switchTo().frame(twitterFrame);

1. Once we have located the <iframe> and have set the focus on it, we can manipulate its elements (we will explain this subject in an upcoming chapter):

WebElement button = driver.findElement(By.id("followbutton"));  
button.click();

1. Compile and execute the script.

### Managing Windows

Web applications usually perform on a single window page. However, during execution, some new windows might be necessary to complete the process. The WebDriver class, combined with the WebDriver.TargetLocator interface, allows automation scripts to manage windows through the following methods:

* java.lang.String getWindowHandle(): Returns a handle to the current window that identifies it within the driver instance being used. It is u Useful for switching to switch to that window in later steps.
* java.util.Set<java.lang.String> getWindowHandles(): Returns a set of handles to the windows. It is useful for switching between windows.

*See the Managing Alerts section for to see a detailed description of the switchTo() method and WebDriver.TargetLocatorinterface.*

#### Managing Windows in an Automation Script

Here, we'll be creating an automation script that manages windows. The steps for completion of this process are as follows:

1. Review and analyze the structure of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_2/exercise02_concept_05.html> file.
2. Create a new Java file for the automation script using IntelliJ IDEA. Make sure that you include the required libraries for the script to work.
3. Using the driver.getWindowHandle() method, save the handle of the parent window in a String variable:

String parentWindowHandle = driver.getWindowHandle();

1. Using the driver.switchTo().window(""); method, switch the focus to the TwitterWindow:

driver.switchTo().window("TwiterWindow");

1. With the focus on "TwitterWindow", use the driver.getTitle() method to verify that the title of the page is Frame Twitter:

if (driver.getTitle().equalsIgnoreCase("Frame Twitter"))

1. Using the System.out.println method, display a message that indicates whether the focus is on the TwitterWindow:

{  
 System.out.println("The script worked, the  
window title is Frame Twitter");  
 } else {  
 System.out.println("Something went wrong,  
the window title is NOT Frame Twitter");  
 }

1. Using the driver.close() method, close "TwitterWindow":

driver.close();

1. Return the focus to the parent window by using the windowHandle that you received on *step 1*.
2. With the focus on the parent window, use the driver.getTitle() method to verify that the title of the page is **Lesson 2**.
3. Compile and run the script.

You should now have a better understanding of how to navigate, resize, and switch between windows, and also how to handle different interactions with alerts.

### Activity: Resizing and Moving Windows with a Selenium Automation Script

**Before You Begin**

* Make sure that you have followed the steps in the previous activity of this chapter.
* Continue to work on the same file from the previous activity of this chapter.

**Scenario**

In *An Overview of Frequently Used Methods*, we used some of the WebDriver methods to start and finalize the script. In this activity, we will work on resizing and moving windows as a part of the test code.

**Aim**

To work on resizing and moving windows using a Selenium automation script.

**Steps for Completion**

1. Using the driver.get() method in the ActivityLesson02 constructor, load [https://www.packtpub.com/.](https://www.packtpub.com/)
2. Using the manage().window().setSize(Dimension) method, resize the current window.
3. Using the manage().window().getSize() method, verify that the size of the window is equal to the size of the one that you set in *step 2*.
4. Using the System.out.println method, display messages that indicate whether the verification was successful.
5. Using the manage().window().maximize() method, maximize the current window.
6. Using the manage().window().getSize() method, verify that the size of the window is different from the one that you set in step 2.
7. Using the System.out.println method, display messages that indicate whether the verification was successful.
8. Using the driver.switchTo().window(""); method, switch the focus to another window.
9. Using the System.out.println method, display a message that indicates whether the focus is on the new window.
10. Using the driver.close() method, close the new window.
11. Compile and run the script.

*To refer to the detailed steps, go to the Solutions section at the end of this book on Page no. 163.*

## Summary

Now that you've completed this chapter, you should have a better understanding of the most frequently used methods of the WebDriver class. You have instantiated a WebDriver for Google Chrome and have looked at how to navigate, resize, and switch between windows.

In the next chapter, you'll work with browser developer tools, learning how to find and interact with elements.

# WebElement Functionality

In the previous chapter, we explored the methods that are frequently used when using the WebDriver class. Additionally, we created a WebDriver instance and controlled the Google Chrome browser momentarily by navigating, switching windows, and resizing it.

Under the hood, a car engine performs heavy processing tasks to keep the automobile running. The driver, however, is unaware of all of these processes, and they are only presented with a dashboard that provides all the necessary information to operate the vehicle.

In a similar way, a web application runs in servers where most of the complex processing takes place: interactions with databases and external services, validation of business logic, and encryption and security measures. The user does not need to know the details of how the application is being executed in the background. Through a web browser, users are only presented with the necessary pages, windows, and forms for them to interact in a (hopefully) transparent and easy manner.

In this chapter, you will learn how to use browser developer tools to review the code behind a web page and how to find and interact with elements to later write and perform tests on them.

By the end of this chapter, you will be able to:

* Use Chrome's Developer Tools
* Explain WebElement functionality
* Work with different techniques to locate elements on a page and learn how to interact with them

## Using Browser Developer Tools Chrome

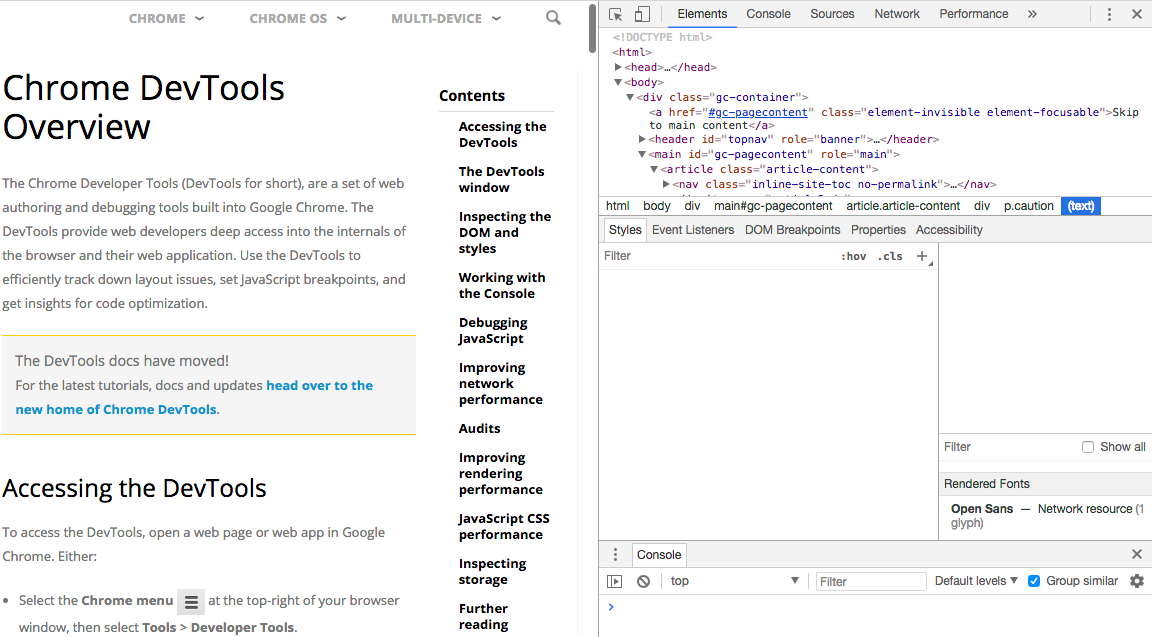
For the user/web application interaction to happen, a web browser is required. The browser presents the user with a frontend that is basically a combination of HTML (Hypertext Markup Language: the one that defines the structure and some behavior of the page), CSS (Cascading Style Sheets: the one that defines the look-and-feel of the page), and JavaScript (the one that allows behavior and interaction beyond HTML capabilities).

It is useful to be able to review and analyze the code behind a web page in order to understand it and define how to locate and interact with its elements for testing purposes. Every browser available provides different tools. As we will only be using Chrome throughout the remainder of this book, we will focus on Chrome Developer Tools.

To open Chrome's Developer Tools, perform these steps:

* Select the Chrome menu.
* Navigate to More tools | Developer Tools.

A new window should open at the bottom or side of your browser:



With Chrome's Developer Tools, we are able to:

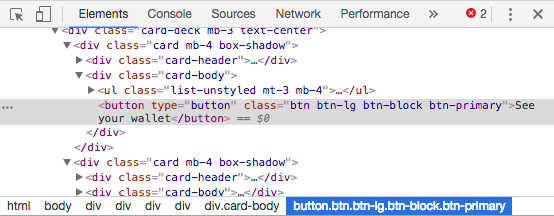
* Get an overview of a web page and its source code at the same time
* Get an overview of the styles applied to a web page, its resources, and the JavaScript code
* Select any element of a web page (a word, a paragraph, a button, a list, and so on), and Chrome's DevTools will highlight the corresponding piece of code
* Try changes on the source code and see how they modify the web page
* Use more advanced tools such as audits, and network and performance

We are now able to understand how to use Chrome Developer Tools.

### Inspecting a Web Page with Chrome Devtools

Given a web page, we will examine its structure and elements using Chrome's DevTools. The aim here is to review the structure and elements of a web page using Chrome's DevTools. The steps for completion of this process are as follows:

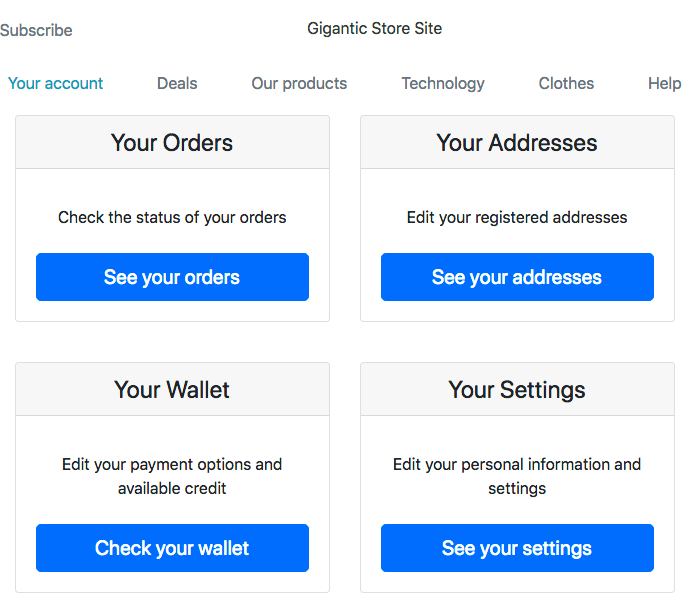
1. Open the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/activity_3_A-1.html>file on Chrome.
2. Open Chrome's DevTools.
3. Right-click on the See your wallet button and choose the Inspect option. You will see a portion of code highlighted on the Elements window to your right:



1. On the Elements window, right-click on the highlighted code over the text of the button and select Edit text.

*If the Edit text note does not appear, make sure that you are right-clicking over See your wallet text.*

1. Change the text of the button to Check your wallet. Verify that the text has changed on the web page to your left:



## Overview of WebElement Functionality

The Selenium WebElement interface allows you to create variables of the WebElement type, which represent HTML elements on a web page. It also provides methods to interact with these elements.

Sometimes, elements on a web page can be dynamically modified, and before interacting with the element, all methods from the WebElement interface will check if the element's reference is still valid before performing any action on it. If the element's reference is not valid, the method will return a StaleElementReferenceExceptionexception.

*Updated information of the WebElement interface can always be found at*[*https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/WebElement.html*](https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/WebElement.html)*.*

Here are the methods with their descriptions:

* void clear(): This clears the value of a text field.
* void click(): This performs a click on an element. It can only be used on visible elements with a width and height bigger than zero (0). If by clicking an element a new page is loaded, all previous references to the element will be invalid.
* WebElement findElement(By by): This receives search parameters and returns the first matching element on the current page, or the NoSuchElementException if no element is found. We will develop this subject in detail in upcoming chapters.
* java.util.List<WebElement> findElements(By by): This receives search parameters and returns all of the matching elements on the current page or an empty list if no element is found. We will develop this subject in detail in upcoming chapters.
* java.lang.String getAttribute(java.lang.String name): Gets the current value of a given attribute of an element.
* java.lang.String getCssValue(java.lang.String propertyName): This gets the value of a given CSS property.
* Point getLocation(): This returns where on the page the top left-hand corner of an element is. Point is a set of (int x, int y) coordinates.
* Rectangle getRect(): This returns the location and size of an element. Rectangle is a set of (int height, int width, int x, int y) measures and coordinates.
* Dimension getSize(): This returns the size of an element. Dimension is a set of (int height, int width) measures.
* java.lang.String getTagName(): This returns the tag name of an element. For the <div class="card-body">Sample text</div> element, this method will return div.
* java.lang.String getText(): This returns the visible inner text of an element. If the element has subelements, it will return a string with no spaces.
* boolean isDisplayed(): This indicates if an element is visible or not.
* boolean isSelected(): This indicates if an element is selected or not. It applies to input elements such as checkboxes, options in a select button, or in a radio button.
* boolean isEnabled(): This indicates if an element is enabled or not.
* void sendKeys(java.lang.CharSequence… keysToSend): This simulates typing into an element.
* void submit(): This submits the current form.

We now have a better understanding of the most frequently used methods of the WebElement class.

### Handling the StaleElementReferenceException

A stale element reference exception is thrown when an element has been deleted or is no longer attached to the DOM. The first case is the most common and it occurs when the page has been refreshed or if the user has navigated to another page.

Here, we will provide you with a sample login page and an automation script that interacts with the email input field. The login page also includes an option for Spanish speakers: when clicking on it, the user is redirected to a Spanish version of the login page (a different HTML file). As we have learned from this chapter, if after clicking on an element a redirection takes place, all previous references to the elements are no longer valid, and a StaleElementReferenceException will be thrown. In this section, we will learn one way to avoid this exception.

The aim here is to avoid and handle the StaleElementReferenceException. The steps for completion of this process are as follows:

1. Open Chrome and go to <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/activity_3_B-1.html>.
2. Use IntelliJ IDEA and copy the contents of <https://github.com/TrainingByPackt/Beginning-Selenium/blob/master/docs/lesson_3/src/main/java/com/beginningselenium/selenium/ActivityB1Lesson03.java> into it.
3. Compile the test and verify that a StaleElementReferenceException is thrown.
4. Option 1: Modify the order of the script. In order to avoid the StaleElementReferenceException, the script should first click on the Spanish version of the login page and then perform the following steps:

public void gettingStaleElementReferenceException() {  
WebElement spanish = driver.findElement(By.  
id("spanish"));  
spanish.click();  
WebElement email = driver.findElement(By.  
id("inputEmail"));  
email.sendKeys("email@gmail.com");  
// following tasks

In this option, we first navigate to the page we want to interact with and then we find the element. The original approach was to find the element first and then navigate to the page, which is why we bumped into the exception.

1. Option 2: Wrap the interaction with the elements in a Java try-catch clause, where we fetch the element again when the StaleElementReferenceException is thrown. It would look like this:

public void gettingStaleElementReferenceException() {  
 // previous code  
 int tries = 0;  
 while (tries < 2) {  
 try {  
 email.sendKeys("email@gmail.com");  
 // We use getAttribute("value") because it is an  
input element, not a text box  
  
if (email.getAttribute("value").equalsIgnoreCase("email@  
gmail.com")) {  
 System.out.println("Script worked, 'email@  
gmail.com' was typed.");  
 } else {  
 System.out.println("Something went wrong with  
the script, 'email@gmail.com' was not set in the email  
box.");  
 }  
 break;  
 } catch (StaleElementReferenceException e) {  
email = driver.findElement(By.id("inputEmail"));  
 }  
 tries++;  
 }  
// following tasks

In this case, we don't alter the initial script but just retry when the exception is found. This concept is more suitable for real websites where elements are removed and placed again in the background, something that is common in progressive web applications.

1. Save the changes, compile the test, and verify that the StaleElementReferenceException is not thrown.

## Interacting with Elements on a Page

A web application consists of many elements that a user can interact with. This includes text boxes, text areas, dropdown lists, lists of items, radio buttons, buttons, checkboxes, and so on. A Selenium automation script has to be able to simulate interactions with these kinds of elements and does so by methods provided by the WebElement class.

### Interacting with Textboxes and Textareas Elements

The following are some of the methods provided by the WebElement class to interact with textboxes and textareas elements during the execution of a script:

* boolean isEnabled(): Indicates if an element is enabled or not.
* java.lang.String getAttribute(java.lang.String name): Gets the current value of a given attribute of an element. This is very useful for getting the contents of text areas, text boxes, and input elements.
* java.lang.String getText(): Returns the visible inner text of an element. If the element has subelements, it will return a string with no spaces.
* void sendKeys(java.lang.CharSequence… keysToSend): Simulates typing into an element.
* boolean isDisplayed(): Indicates if an element is visible or not.

#### Interacting with Textboxes and Textareas During an Automation Script

The aim here is to create an automation script that interacts with textboxes and textareas. We will assume that the textbox/textarea element has been found by means of its ID (we will go deeper into the findElement method in an upcoming section):

WebElement textArea = driver.findElement(By.  
id("aboutYourself"));

Open the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/exercise_3_1.html> file and use IntelliJ IDEA for the creation of a Selenium script.

The steps for completion of this process are as follows:

1. Review and analyze the structure and behavior of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/exercise_3_1.html>file.
2. Create a new Java file for the automation script. Make sure that you include the required libraries:

package com.beginningselenium.selenium;  
  
import org.openqa.selenium.By;  
import org.openqa.selenium.StaleElementReferenceException;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.chrome.ChromeDriver;

1. Locate the textarea of the By.id "aboutYourself":

WebElement textArea = driver.findElement(By.  
id("aboutYourself"));

1. Verify that the textarea is enabled and visible:

if (textArea.isEnabled() && textArea.isDisplayed())

1. Using the System.out.println method in the if loop, display messages indicating whether the verification was successful or not:

System.out.println("The text area is visible and  
displayed");  
else {  
 System.out.println("Something went wrong, the text  
area is not visible and displayed");  
 }

1. If the textarea is enabled and visible, verify that it is empty:

// Checking for an empty text area  
if ("".equals(textArea.getAttribute("value")))

*Have the code of steps 6-11 inside the if (textArea.isEnabled() && textArea.isDisplayed()) loop.*

1. Using the System.out.println method, display messages indicating whether the verification was successful or not:

{  
 System.out.println("The text area is empty");  
 } else {  
 System.out.println("Something is wrong, the text  
area NOT empty");  
 }

1. Input some text and then verify that the text was actually typed:

textArea.sendKeys("This is a sample text.");  
if ("This is a sample text.".equals(textArea.  
getAttribute("value")))

1. Using the System.out.println method, display messages indicating whether the verification was successful or not and that the name was sent.

{  
 System.out.println("Text was correctly typed into  
the text area.");  
 } else {  
 System.out.println("Something went wrong, text  
was not typed into the text area.");  
 }

1. Clear the textarea contents and verify that it is empty:

textArea.clear();  
if ("".equals(textArea.getAttribute("value")))

1. Using the System.out.println method, display messages indicating whether the verification was successful or not and that the name was sent:

{  
 System.out.println("The text area is empty after  
cleaning it though a Selenium command");  
 } else {  
 System.out.println("Something went wrong, the  
text area was not cleaned");  
 }

1. Compile and run the script.

### Interacting with Dropdown and Lists

Dropdown and lists elements are manipulated by means of the Select class. This class provides methods and properties to interact with dropdown and lists that are created with the <select> element:

* void deselectAll(): Clears all selected entries
* void deselectByIndex(int index): Clears the option at the given index
* void deselectByValue(java.lang.String value): Clears all options that match a given value
* void deselectByVisibleText(java.lang.String value): Clears all options that display text matching a given value
* java.util.List<WebElement> getAllSelectedOptions(): Gets all selected options of the list
* WebElement getFirstSelectedOption(): Gets the first selected option of the list
* java.util.List<WebElement> getOptions(): Gets all options of the list
* boolean isMultiple(): Indicates if the list supports multiple options at the same time or not
* void selectByIndex(int index): Selects the option at the given index
* void selectByValue(java.lang.String value): Selects all options that match a given value
* void selectByVisibleText(java.lang.String text): Selects all options that display text matching a given text

*Up-to-date documentation of the Select class can be found at*[*https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/support/ui/Select.html*](https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/support/ui/Select.html)*.*

#### Interacting with Dropdown and Lists During an Automation Script

The aim here is to create an automation script that interacts with dropdown and lists.

For the following example, we will assume that the dropdown element has been found by means of a combination of the Select method and its ID (we will go deeper into the findElement method in an upcoming section):

Select list = new Select(driver.findElement(By.  
id("monthOfBirth")));

Open the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/exercise_3_1.html> file and use IntelliJ IDEA for the creation of a Selenium script. The steps for completion of this process are as follows:

1. Review and analyze the structure and behavior of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/exercise_3_1.html> file.
2. Create a new Java file for the automation script. Make sure that you include the required libraries:

package com.beginningselenium.selenium;  
  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.chrome.ChromeDriver;  
import org.openqa.selenium.support.ui.Select;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;

1. First, we will work with a single choice list. Locate the dropdown list of By.id "monthOfBirth":

Select singleChoiceList = new Select(driver.findElement(By.id("monthOfBirth")));

1. If the dropdown is enabled and visible, verify that it does not allow multiple selections and that it contains 13 options (including "Choose…"):

if (!singleChoiceList.isMultiple() && singleChoiceList.  
getOptions().size() == 13)  
{  
}

1. Using the System.out.println method, display messages indicating whether the verification was successful or not:

System.out.println("The list does not accept multiple  
choices and contains 13 options (including 'Choose...').");

1. If the list does not allow multiple selection and its size is 13, select the option "February" by sending its value:

if (!singleChoiceList.isMultiple() && singleChoiceList.  
getOptions().size() == 13)  
{  
singleChoiceList.selectByVisibleText("February");  
}

You can also select an option by value:

singleChoiceList.selectByValue("february");

Or by index:

singleChoiceList.selectByIndex(2);

1. Verify that "February" is selected as an option and use the System.out.printlnmethod, which displays messages indicating whether the verification was successful or not and the option chosen:

if (!singleChoiceList.isMultiple() && singleChoiceList.  
getOptions().size() == 13)  
 {  
 singleChoiceList.selectByVisibleText("February");  
 if (singleChoiceList.getFirstSelectedOption().  
getText().equalsIgnoreCase("February"))  
 {  
 } else {  
 }  
}

1. Now, we will work with a multiple-choice list. Locate the dropdown list of By.id "monthOfBirth":

Select multipleChoiceList = new Select(driver.  
findElement(By.id("hobbies")));

1. If the dropdown is enabled and visible, verify that it does allow for multiple selections and that it contains 4 options:

if (multipleChoiceList.isMultiple() && multipleChoiceList.  
getOptions().size() == 4)  
 {  
 }

1. Using the System.out.println method, display messages indicating whether the verification was successful or not.

System.out.println("The list does accept multiple choices  
and contains 4 options.");

1. If the list does allow multiple selection and its size is 4, select the different options by sending its values:

if (multipleChoiceList.isMultiple() && multipleChoiceList.  
getOptions().size() == 4)  
{  
 multipleChoiceList.selectByVisibleText("Reading");  
 multipleChoiceList.selectByVisibleText("Sports");  
 multipleChoiceList.selectByVisibleText("Traveling");  
}

1. Deselect an option using the value attribute:

multipleChoiceList.deSelectByValue("sports");

You can also deselect an option by its index:

multipleChoiceList.deselectByIndex(0);

Or by its visible text:

multipleChoiceList.deselectByVisibleText("Sports");

1. Verify the number of choices selected and use the method to display messages indicating the number of options chosen:

if (multipleChoiceList.getAllSelectedOptions().size() == 2)  
{  
 System.out.println("It worked, 2 options have been  
chosen");  
}

1. Verify that the two options are actually selected:

List<String> expectedSelection = Arrays.asList("Reading",  
"Traveling");  
List<String> actualSelection = new ArrayList<String>();  
for (WebElement element : multipleChoiceList.  
getAllSelectedOptions()) {  
 actualSelection.add(element.getText());  
}  
if (actualSelection.containsAll(expectedSelection)) {  
} else {  
}

1. Compile and run the script.

### Interacting with Radio Buttons and Radio Button Groups

Radio buttons are commonly used in web applications to offer the user a way to select options that are mutually exclusive. WebDriver offers support for radio buttons and radio groups through the WebElement class. Here are some of the methods provided so that you can interact with radio buttons and radio buttons groups elements during the execution of a script:

* boolean isSelected(): Indicates if an element is selected or not. It applies to input elements such as checkboxes, options in a select button, or in a radio button.
* void click(): Performs a click on an element. It can only be used on visible elements with a width and height bigger than zero (0). If by clicking an element a new page is loaded, all previous references to the element will be invalid.

#### Interacting with Radio Buttons and Radio Buttons Groups During an Automation Script

The aim is to create an automation script that interacts with radio buttons and radio buttons groups. For the following example, we will assume that the radio button element has been found by means of the findElement method (we will go into more detail about this method in an upcoming section):

WebElement masters = driver.findElement(By.  
cssSelector("input[value='masters']"));

Open the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/exercise_3_1.html> file and use IntelliJ IDEA for the creation of a Selenium script. The steps for completion of this process are as follows:

1. Review and analyze the structure and behavior of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/exercise_3_1.html>file.
2. Create a new Java file for the automation script. Make sure that you include the required libraries.

package com.beginningselenium.selenium;  
  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.chrome.ChromeDriver;

1. Locate the radio button "Masters":

WebElement masters = driver.findElement(By.  
cssSelector("input[value='masters']"));

If we know that the radio button we want to work with belongs to a group, we can locate the group by its name:

List<WebElement> degreeLevel = driver.findElements(By.  
name("degree"));  
And iterate through its elements:  
for (WebElement degree : degreeLevel)  
{  
 if(degree.getAttribute("value").equals("masters"))  
 {  
 if(!degree.isSelected()) {  
 type.click();  
 }  
 break;  
 }  
}

1. Verify that the radio button "Masters" is enabled and visible:

if (masters.isEnabled() && masters.isDisplayed())  
{  
  
}

1. Using the System.out.println method, display messages indicating whether the verification was successful or not:

System.out.println("The radio button is enabled and  
visible.");

1. If the radio button is enabled and visible, verify that it is not selected before clicking on it:

if (masters.isEnabled() && masters.isDisplayed())  
{  
 if (!masters.isSelected())  
 {  
 masters.click();  
 }  
}

We can select a radio button by clicking on it after it has been selected.

1. Verify that "Masters" has been selected and use the System.out.println method to display messages, indicating whether the verification was successful or not and the option chosen:

if (masters.isEnabled() && masters.isDisplayed())  
{  
 if (!masters.isSelected())  
 {  
 masters.click();  
 if (masters.isSelected())  
 System.out.println("It worked, the 'Masters'  
option was selected");  
 else  
System.out.println("Something went wrong,  
'Masters' was not selected.");  
 }  
}

1. Compile and run the script.

### Interacting with Checkboxes

Checkboxes are widely used in web applications, mostly in situations where the user needs to select one or more available options, for example, when selecting user hobbies in a registration form. WebDriver offers support for checkboxes through the WebElement class.

Here are some of the methods provided so that you can interact with checkboxes groups elements during the execution of a script:

* boolean isSelected(): Indicates if an element is selected or not. It applies to input elements such as checkboxes, options in a select button, or in a radio button.
* void click(): Performs a click on an element. It can only be used on visible elements with a width and height bigger than zero (0). If by clicking an element a new page is loaded, all previous references to the element will be invalid.

#### Interacting with Checkboxes During an Automation Script

The aim is to create an automation script that interacts with checkboxes. For the following example, we will assume that the checkbox element has been found by means of the findElement method (we will go into more detail about this method in an upcoming section):

WebElement receiveEmails = driver.findElement(By.  
id("emailUpdates"));

Open the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/exercise_3_1.html> file and use IntelliJ IDEA for the creation of a Selenium script.

1. Review and analyze the structure and behavior of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/exercise_3_1.html>file.
2. Create a new Java file for the automation script. Make sure that you include the required libraries.
3. Locate the checkbox "emailUpdates".

WebElement emailUpdates = driver.findElement(By.  
id("emailUpdates"));

1. Verify that the checkbox is enabled and visible:

if (emailUpdates.isEnabled() && emailUpdates.isDisplayed())  
{  
  
}

1. Using the System.out.println method, display messages indicating whether the verification was successful or not.
2. If the checkbox is enabled and visible, verify that it is not selected before clicking on it:

if (emailUpdates.isEnabled() && emailUpdates.isDisplayed())  
{  
 if (!emailUpdates.isSelected())  
 {  
 emailUpdates.click();  
 }  
}

We can deselect a checkbox by clicking on it after it has been selected.

1. Verify that the checkbox is selected and use the System.out.println method to display messages, indicating whether the verification was successful or not and that the checkbox has been checked:

if (receiveEmails.isEnabled() && receiveEmails.  
isDisplayed())  
{  
 if (!emailUpdates.isSelected())  
 {  
 emailUpdates.click();  
 if (emailUpdates.isSelected())  
 System.out.println("Load of the test worked,  
checkbox has been selected");  
 else  
 System.out.println("Something went wrong with  
the test, checkbox has not been selected");  
 }  
}

1. Compile and run the script.

We now have a better understanding of how to interact with the elements of a web page.

### Activity: Filling in a Form and Submitting it

**Scenario**

In the different concepts of this section, we have worked on interacting with many types of web elements that belong to a form. In this activity, we'll make use of the different concepts to create an automation script that interacts with the different types of elements on a web page.

**Aim**

To create a full automation script that interacts with different types of elements on a web page.

**Steps for Completion**

1. Review and analyze the structure and behavior of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/exercise_3_1.html>file.
2. Create a new Java file for the automation script. Make sure that you include the required libraries.
3. Locate the textbox of id "firstName", and input some text. In this case, input "John".
4. Locate the textbox of id "lastName", and input some text ("Doe").
5. Locate the dropdown list of id "dayOfBirth"; select the option "20".
6. Locate the dropdown list of id "monthOfBirth"; select the option "March" by sending its value.
7. Locate the dropdown list of id "yearOfBirth"; select the option "1990" by sending its value.
8. Locate the dropdown list of id "hobbies"; select these two options: Reading and Sports.
9. Locate the radio button "Masters"; select a radio button by clicking on it after it has been selected.
10. Locate the checkbox "emailUpdates". and enable the "I want to receive email updates" field.
11. Locate the textarea of id "aboutYourself", and input some text.
12. Include an instruction to click on the Submit button.
13. Compile and run the script.

*To refer to the detailed steps, go to the Solutions section at the end of this book on Page no. 165.*

## Element Locator Types ID, Names, XPath, CSS, and So On

Locating elements on a web page is done by using the findElement() and findElements()methods.

The findElement() method:

* Returns a WebElement based on a given search criteria
* Returns the first matching element
* Throws a NoSuchElementFound exception when the element cannot be found.

The findElements() method:

* Returns a list of WebElements based on a given search criteria
* Returns an empty list if no elements were found

The previously mentioned search criteria consist of a set of locator types:

* By ID: Finds elements by the value of their ID attribute. It is the most common way to identify elements on a page. However, not all elements have an ID, or they might be dynamically generated.  
  Here's the syntax:

driver.findElement(By.id(<element ID>))

Here's an example:

WebElement username = driver.findElement(By.id("username"));

* By Name: Finds elements by the value of their name attribute.  
  Here's the syntax:

driver.findElement(By.name(<element name>))

Here's an example:

WebElement password = driver.findElement(By.  
name("password"));

* By Class name: Finds elements by the value of their class attribute.  
  Here's the syntax:

driver.findElement(By.className(<element class>))

Here's an example:

WebElement username = driver.findElement(By.  
className("username"));

* By Tag Name: Finds elements by the value of their HTML tag. This method is not recommended if one is trying to locate a specific element on a page with multiple instances of such elements.  
  Here's the syntax:

driver.findElement(By.tagName(<htmlTagName>))

Here's an example:

// Locates a table by its ID  
WebElement table = driver.findElement(By.id("table"));  
  
// Counts the row of the table by locating the <tr>  
elements of the table  
List<WebElement> rows = table.findElements(By.  
tagName("tr"));  
int totalRows = rows.size();

* By Link Text: Finds a link by its displayed text. It returns all matching links that meet the specified text.  
  Here's the syntax:

driver.findElement(By.link-Text(<linktext>))

Here's an example:

//Locates the link corresponding to the HELP option  
WebElement helpLink = driver.findElement(By.  
linkText("HELP"));  
  
// Takes the URL out of the link  
string link = helpLink.getAttribute("href"));

* By Partial Link Text: Finds a link by a partial text. It is useful for links that are dynamically created. It returns all matching links that meet the specified text.  
  Here's the syntax:

driver.findElement(By.partialLinkText(<linkText>))

Here's an example:

// The "Messages" link includes the number of unread  
messages and it is generated dynamically  
WebElement msgLink = driver.findElement(By.  
partialLinkText("Messages"));  
  
// Takes the URL out of the link  
String linkText = msgLink.getText();

* By XPath: Finds elements via XPath queries. With this method, it is possible to locate a parent element using a child element and vice versa.

Here's the syntax:

driver.findElement(By.xpath(<xpathexpresion>))

Here's an example:

// Locates an element with an absolute path  
// If the structure of the HTML changes, the locator will  
not find the element  
WebElement email = driver.findElement(By.xpath("/html/body/  
div/form/input"));  
  
// Locates an element with a relative path assuming that  
the email textbox is the first first <input> element of the  
HTML page  
WebElement email = driver.findElement(By.xpath("//input"));  
  
// Locates an element using predicates assuming that the  
email textbox is the third <input> element of the HTML page  
WebElement email = driver.findElement(By.xpath("//  
input[3]"));  
  
// Locates an element using attributes values of an element  
WebElement locatorsDiagram = driver.findElement(By.  
xpath("//img[@alt='Selenium Locators Diagram']"));  
  
// Locates an element by combining attributes values  
WebElement button = driver.findElement(By.xpath("//input[@  
type='submit'][@value='accept']"));  
  
// Locates all elements that do not have an specific  
attribute specified  
List<WebElement> imagesWithAlt = driver.findElements(By.  
xpath ("//img[not(@alt)]"));

* By CSS Selector: Finds elements by their CSS selector.  
  Here's the syntax:

driver.findElement(By.cssSelector(<selector>))

Here's an example:

/ Locates an element with an absolute path  
// If the structure of the HTML changes, the locator will  
not find the element  
WebElement email = driver.findElement(By.cssSelector("html  
body div form input"));  
  
// Locates an element with a relative path assuming that  
the email textbox is the first <input> element of the HTML  
page  
WebElement email = driver.findElement(By.  
cssSelector("input"));  
  
// Locates an element by its class attribute, specifying  
the type of HTML tag (input) and the value of the class  
attribute (accept)  
WebElement acceptButton = driver.findElement(By.  
cssSelector("input.accept"));  
  
// Locates an element using the ID attribute by specifying  
the type of HTML tag (input) and the value of the ID  
attribute (email):  
WebElement email = driver.findElement(By.  
cssSelector("input#email"));  
  
// Locates an element by the name attribute  
WebElement userName = driver.findElement(By.  
cssSelector("input[name=username]"));

We now have a better understanding of how to locate elements on a page using the findElement() and findElements() methods.

### Activity: Locating Elements

**Scenario**

For a single web page, we will write locators using the different techniques mentioned in this section. For a single web page, we'll locate elements by ID, name, class name, HTML tag, link, xpath, and CSS.

**Aim**

To create an automation script that uses different kinds of locators to find elements on a web page.

**Steps for Completion**

1. Open <https://trainingbypackt.github.io/Beginning-Selenium/lesson_3/activity_3_D-1.html> and review it.
2. Using IntelliJ IDEA, import the required packages.
3. Locate an element by using an ID. Use "lastname" as the ID. Print the output when it's displayed and when it's not.
4. Locate an element by using a name. Use "hobbies" as the ID. Print the output when it's displayed and when it's not.
5. Locate an element by using a class name. Use "form-control" as the class name. Print the output when the first name is displayed and when it's not.
6. Locate an element by using an HTML tag. Use "div" as the class name. Print the output when div.size() > 0 and otherwise.
7. Locate an element by using a link. Use "Spanish" as the class name. Print the output when link = spanishLink.getAttribute("href") and when it's not.
8. Locate an element by using xpath. Use "//select" as the xpath. Print the output when dayOfBirth.getOptions().size() > 0 and when this condition is not met.
9. Locate an element by using CSS. Use "#firstName" as the CSS attribute. Print the output when the first name with the CSS is displayed and when it's not.
10. Verify that each element was found by interacting with it.
11. Compile and run the script.

*To refer to the detailed steps, go to the Solutions section at the end of this book on Page no. 167.*

## Summary

In this chapter, you learned how to use Chrome's DevTools. You got an overview of the WebElement functionality. You also worked  with different techniques to locate elements on a page and how to interact with them. With the completion of the first three chapters, you've configured your environment and have got the correct fundamentals for working with the Selenium WebDriver.

In the next chapter, you'll understand the structure of the Document Object Model, and work with search techniques to build complex element locators.

# Advanced Element Location

In the fourth chapter of this book, you will learn how to identify web elements in a web application by using effective element locators. After reviewing the basic concepts and the structure of the **Document Object Model** (**DOM**), we will illustrate how to locate elements within other elements. Finally, we will introduce you to different search techniques that can be useful when building complex element locators.

By the end of this chapter, you will be able to:

* Navigate through the DOM to find elements
* Search for elements that are nested within existing ones
* Create complex locators to find elements that are difficult to reach

## Navigating the DOM

Having a correct and reliable element is one of the most important aspects of the process of automating a web application. Elements allow us to interact with a web application just like a real user would; that is why it is important to understand how  
to locate elements and how to choose good locators.

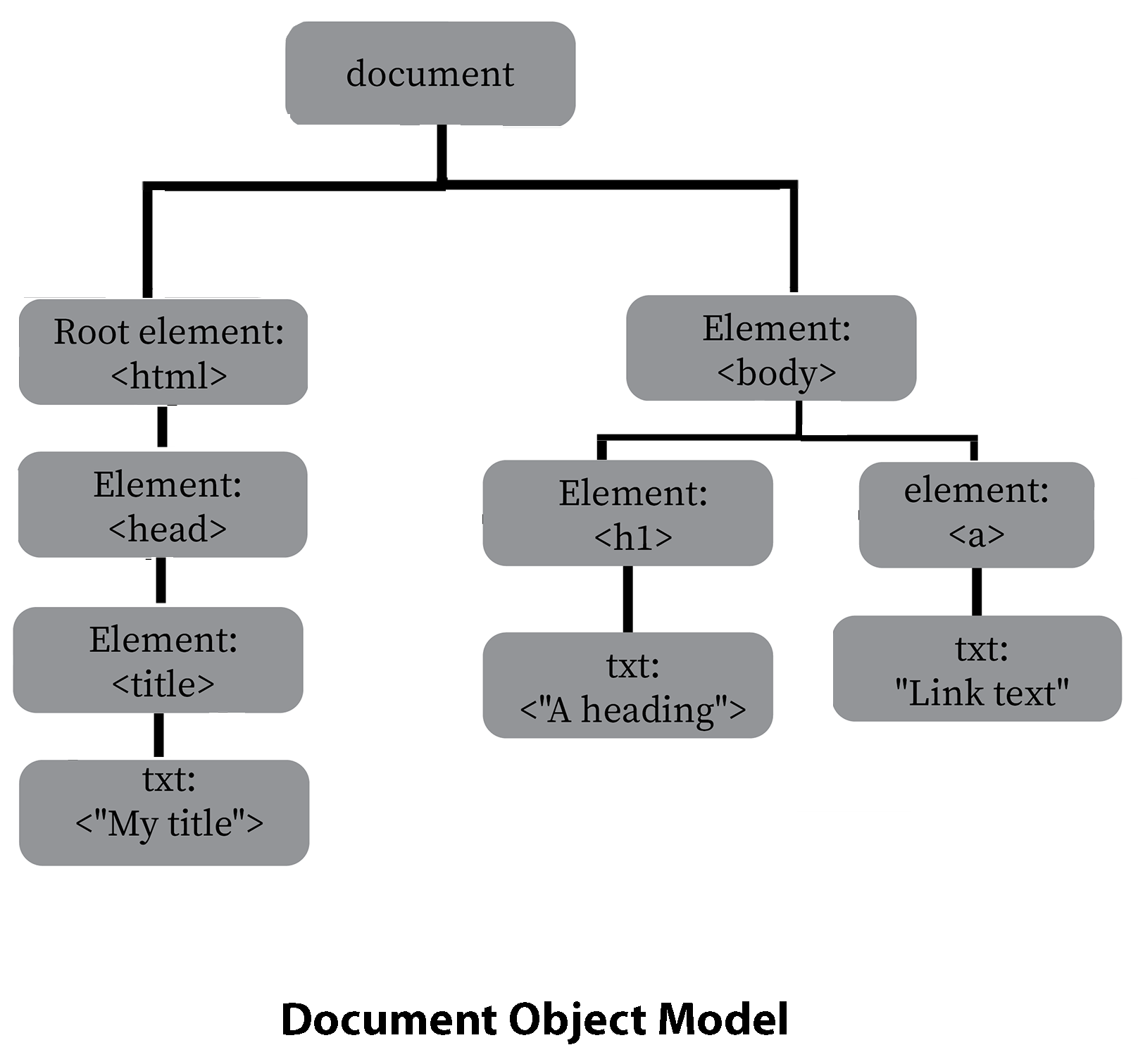
### Understanding the DOM's Structure

A very relevant question you may ask would be if DOM and HTML are the same. This is meant to be a very tricky question, because visually, they look similar. HTML is the code used to build the page; when the browser takes the HTML and renders it, it becomes the DOM.

Two differences between the DOM and the HTML are as follows. If the HTML has an error (for example, if you create an <li> element without the closing </li>), the browser will insert it, making the DOM correct. The second difference is that you can modify the content of the DOM in real time, via JavaScript.

Selenium can only interact with the elements that a real user can interact with. This means that if a real user cannot click or type on an element, Selenium won't be able to either.

Now, we will briefly jump to the DOM's structure. This will help you to learn how elements are placed and how they can be navigated through. Let's analyze the following diagram:



All of the items between tags (< and >) are elements of the DOM, and all of the nested elements become children of the elements enclosing them. Similarly, elements that are next to each other at the same level are considered siblings.

In the preceding diagram, the elements <h1> and <a> are children of the <body> element, which becomes their parent element. In the same vein, the elements <h1> and <a> are siblings; <h1> is the first child, and <a> is the last child.

### The Relationship between Selenium and the DOM

A very good question now would be why it is important to understand the basic structure of the DOM. Selenium can only interact with the same elements that a real user can interact with. This means that if a real user cannot click or type on an element, Selenium won't be able to either. Selenium and the DOM connect to each other through the elements present in the DOM. To create a healthy relationship between the two, elements should be clear and easily identifiable. Nowadays, web applications have a DOM that is generated dynamically (or simply, with a complex structure), and it is not possible to find a good locator without first understanding the DOM's structure. For example, identifying when one element is the child of another easily reachable element can simplify our tasks.

#### Identifying Elements and Creating Locators for Dom Elements

The aim here is to navigate through the DOM by selecting elements with good locators. Consider the following DOM:

<html>  
 <head>  
 <title>About Me</title>  
 </head>  
 <body>  
 <h1 id="about">About Me</h1>  
 <ul id="list-group">  
 <li class="list-group-item">Name: <span id="name">John Doe</  
span></li>  
 <li class="list-group-item">Phone: <span  
id="phone">400-6970</span></li>  
 <li class="list-group-item">Hometown: <span  
id="hometown">Springfield</span></li>  
 </ul>  
 </body>  
</html>

Based on what you have learned in the previous chapters, identify all of the elements inside the <body> tag, and understand their roles (parents, children, or siblings).

Afterwards, create locators for all of the <ul> element's children, each of the <span>elements, and the first child of the <body> element. The steps for completion are as follows:

1. Open <https://trainingbypackt.github.io/Beginning-Selenium/lesson_4/exercise_4_1.html> in Chrome.
2. Open the Chrome DevTools to search for elements.
3. Use the Elements tab in Chrome DevTools to search for the requested elements.
4. Try some different alternatives for locating the same element, and try to stick to the simplest and shortest alternative.  
     
   The "<ul>" element can be either located by the ID, CSS locator, and XPATH locator. After trying the three alternatives for the element, it should be clear that the shortest one will be the easiest one to maintain.

You should now be able to correctly identify the elements present in any DOM. You should also be able to understand how the elements are related to each other, via parent, child, and sibling relationships.

#### Retrieve Information from a Table

The aim here is to interact with the DOM through the Chrome DevTools console, and to work through the process of retrieving elements with different locators. The steps for completion are as follows:

1. Open <https://trainingbypackt.github.io/Beginning-Selenium/lesson_4/activity_4_A-1.html>, and inspect it through the DevTools JavaScript console.
2. Count all of the names and print them in the console:

document.getElementsByClassName('name').length  
> 3

1. Print each of the names, and access each element through its index:

document.getElementsByClassName('name')[0].innerHTML  
> "John"  
document.getElementsByClassName('name')[1].innerHTML  
> "Mary"  
document.getElementsByClassName('name')[2].innerHTML  
> "Peter"

1. Repeat steps 2 and 3 to print each of the last names and emails.

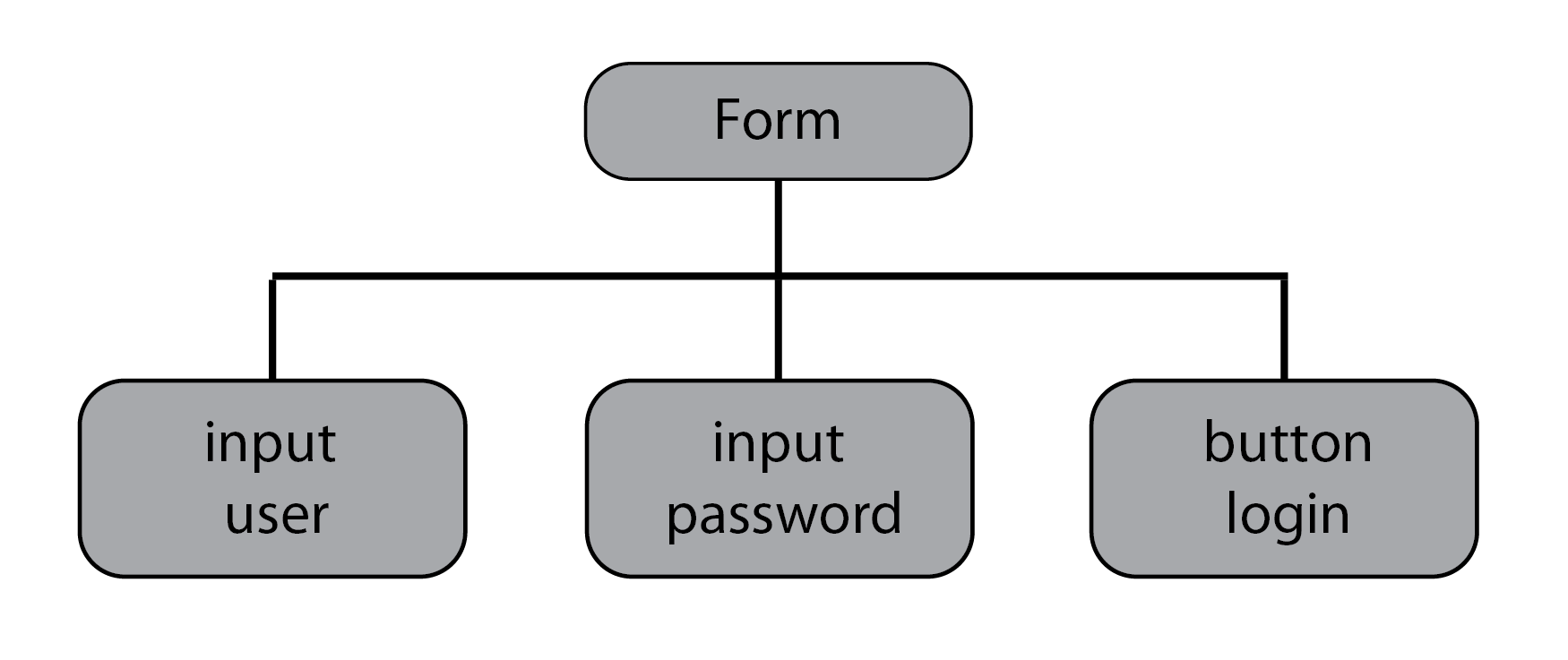
## Searching Within Previously Found Elements

Automating web applications primarily involves finding elements in the DOM. Sometimes, however, we can't figure out the locator for the element we need, either because it is missing attributes, or there are simply too many elements with the same attributes.

Nevertheless, it will frequently happen that we know how to locate the parent or the sibling of the element we need. This is a common scenario in nested elements, such as a list with several items. In these cases, we can first locate the element that we know, and then search for the element that we need.

### Locating Elements Based on Their Relationships

With CSS locators, we can find adjacent elements by combining the parent, child, and sibling relationships:



To reach the password field in the preceding diagram, we can either use the parent-child relationship between the form and password elements or the sibling relationship between the user and password elements.

Using the parent-child relationship, if the form has the ID login, a CSS selector will look as follows:

form#login > input:nth-child(2).

The Java code to use this selector would be as follows:

WebElement user = driver.findElement(By.cssSelector("form#login >  
input:first-child"));

To get the first element, we can use form#login > input:first-child. With :nthchild(n), it is possible to reach any element that we want; we only have to know its index. Similarly, to get the last element's child, we can use :last-child.

In the preceding example, input:nth-child(2) and input:last-child would return the same element, since the password input includes both the second and last child at the same time.

If we wanted to use the sibling relationship, we could first find the user element and then use the + symbol to locate its sibling's password. The CSS selector would look as follows:

form#login > input:first-child + input

The corresponding Java code would look as follows:

WebElement user = driver.findElement(By.cssSelector("form#login >  
input:first-child + input"));

#### Searching for Elements Through Known Elements

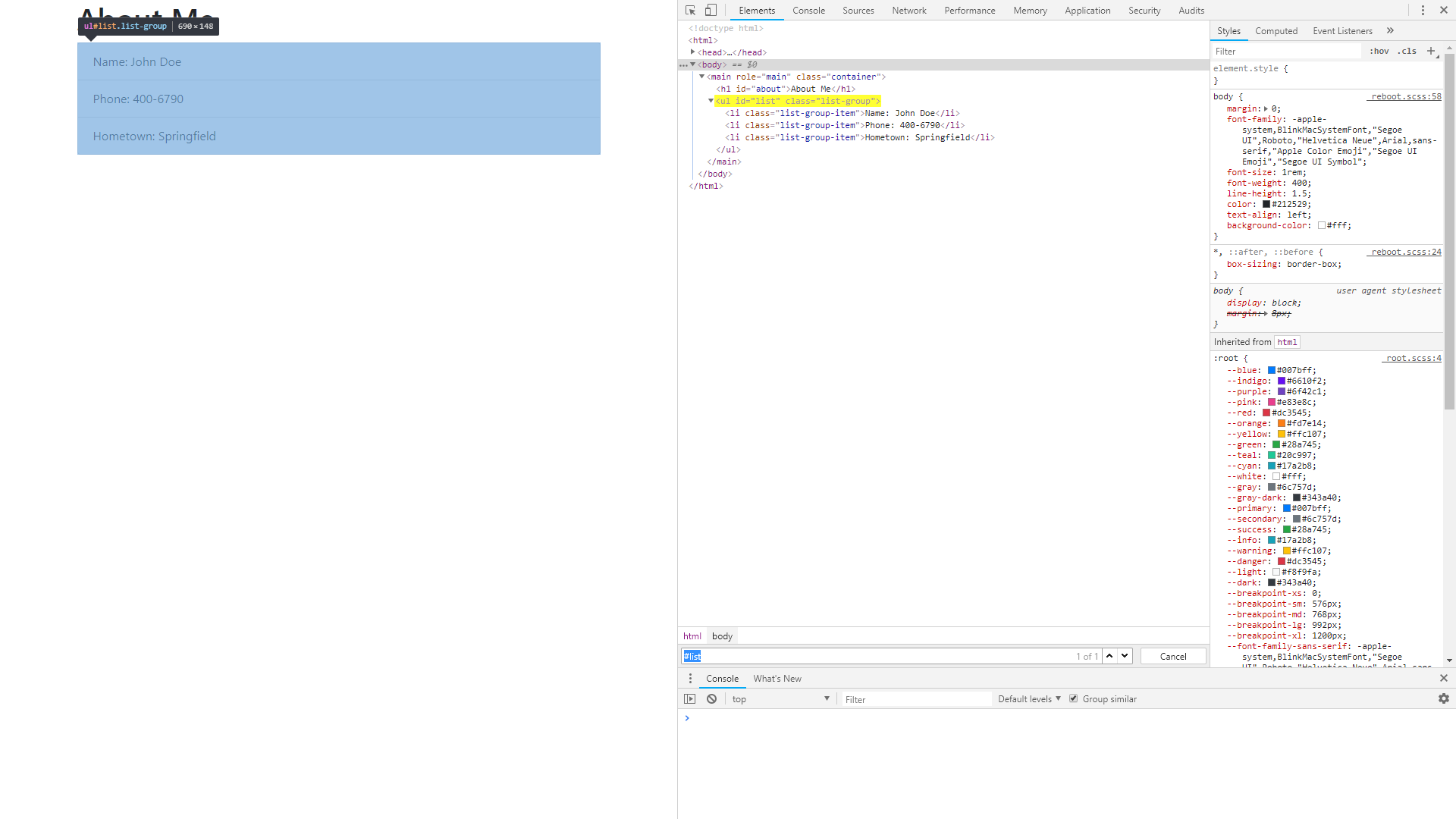
he aim is to generate CSS selectors and use them in automation code, in order to simplify the interaction with elements that are difficult to reach. Consider the following, slightly modified version of the DOM that we used in *Retrieve Information from a Table*:

<html>  
 <head>  
 <title>About Me</title>  
 </head>  
 <body>  
 <h1 id="about">About Me</h1>  
 <ul id="list" class="list-group">  
 <li class="list-group-item">Name: John Doe</li>  
 <li class="list-group-item">Phone: 400-6790</span></li>  
 <li class="list-group-item">Hometown: Springfield</span></  
li>  
 </ul>  
 </body>  
</html>

Note that the list elements no longer have unique IDs. We'll create CSS selectors to find the list elements, and we will use them to implement locators in Java code. You can use either the parent-child relationship or the sibling relationship. The steps for completion are as follows:

1. Open <https://trainingbypackt.github.io/Beginning-Selenium/lesson_4/exercise_4_2.html> in the Chrome browser.
2. Open the Chrome DevTools to try out the selectors indicated in the next steps.
3. To identify the first item in the list (John Doe), use the selector #list to get its parent:

 The following screenshot shows what your screen should look like:



1. Retrieve the first child of #list with li:first-child.
2. Combine the two selectors (#list and li:first-child) to obtain a solid locator for the element that we are looking for.
3. Use li:last-child rather than li:first-child, to reach the last item in the list.
4. Use the nth-child keyword, combined with index 2, to reach the second item in the list. Since we need the second element, it will look as follows: li:nth-child(2). The complete selector will look as follows: #list li:nth-child(2).
5. After all of the selectors have been found, use them to create the locators with Java:

WebElement user = driver.findElement(By.  
cssSelector("form#login > input:first-child + input"));  
WebElement user = driver.findElement(By.  
cssSelector("form#login > input:nth-child(2) + input"));

*Sometimes, the creation of an effective element locator is not straightforward, because the element doesn't have an ID or any distinctive attributes.*

## Creating Complex Locators to Reach Any Element

By now, it is probably clear how important CSS selectors are. They offer great performance, because all major browsers have implemented CSS parsing engines to render website content, using the styles defined in the CSS syntax. Selenium takes advantage of the parsing engine in the browser when a CSS selector is used, and so can you.

### Common Ways to Use CSS Selectors

The two most common ways to use CSS selectors involve either using the class attributes or the element's ID (when available). To use class attributes in a CSS selector, you must list the element type (for example, a button), add a dot, and add the class attribute. If you have a button element with the class big-button, the selector will look as follows: button.big-button.

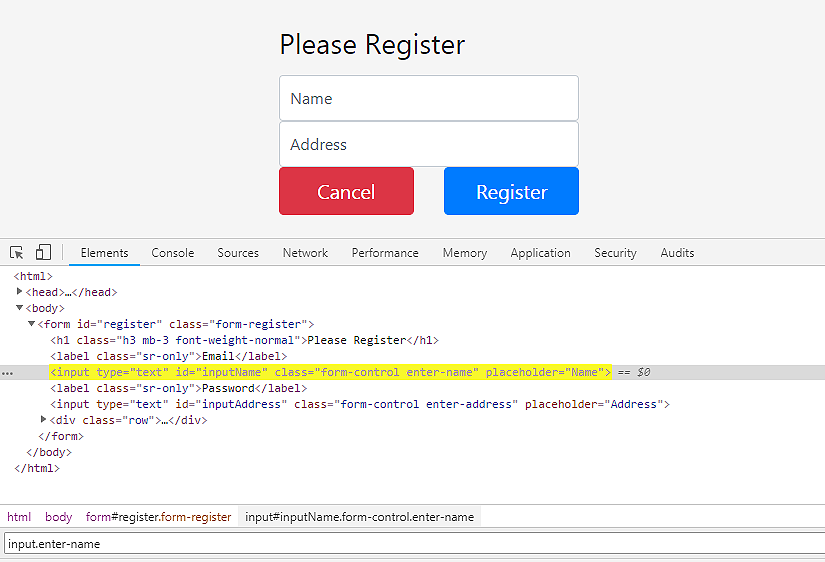
#### Finding Elements by Using Class Attributes and IDs

The aim here is to identify when it is suitable to use class attributes or element IDs in the creation of CSS selectors. The following DOM models a simple registration form:

<html>  
 <head>  
 <title>Register</title>  
 </head>  
 <body>  
 <form id="register" class="form-register">  
 <h1 class="h3 mb-3 font-weight-normal">Please Register</h1>  
 <label for="inputName" class="sr-only">Name</label>  
 <input type="text" id="inputName" class="form-control"  
placeholder="Name">  
 <label for="inputAddress" class="sr-only">Password</label>  
 <input type="text" id="inputAddress" class="form-control"  
placeholder="Address">  
 <div class="row">  
 <div class="col-md-6 mb-3">  
 <button class="btn btn-lg btn-danger btn-block"  
id="cancelButton" type="submit">  
 Cancel  
 </button>  
 </div>  
 <div class="col-md-6 mb-3">  
 <button class="btn btn-lg btn-primary btn-block"  
id="register" type="submit">  
 Register  
 </button>  
 </div>  
 </div>  
 </form>  
 </body>  
</html>

Create CSS selectors for the input elements by using class attributes, and create the register button by using the element ID. Implement locators in Java by using the CSS selectors that are generated. The steps for completion are as follows:

1. Open <https://trainingbypackt.github.io/Beginning-Selenium/lesson_4/exercise_4_3.html> in Chrome.
2. Open the Chrome DevTools.
3. To select the input for the name section, identify the element type (input) and its class attribute (enter-name) its parent.
4. Combine the element type and the class attribute to form the CSS selector; it should look as follow: input.enter-name. This is how your screen should look like:



1. Follow steps 2 and 3 to create a CSS selector for the address section.
2. Note that the element button has the ID register, but, due to a mistake, the element form also has the ID register. In this case, combine the element tag with its ID. The CSS selector will look as follows: button#register.
3. After all of the selectors have been found, use them to create the locators with Java.

### Locating Elements by Their Attributes

In modern web applications, where a lot of the elements are generated dynamically, often, it's not enough to know the element ID or class attribute to build a unique selector. There may be cases wherein an element has the same class, name, or even ID, but they are differentiated by having different values in the attributes.

The syntax of a CSS selector that uses attributes is, in general, very simple, consisting of three basic elements: the element type, the attribute, and the value of the attribute (for example, element[attribute='value']).

It is also possible to combine attributes to create a unique and more robust CSS selector. This is helpful when only the union of two or more attributes can help us to differentiate the element from other elements.

A combined CSS selector looks as follows:

element[attribute1='value1'][attribute2='value2']

Partial matches are an additional alternative, when an exact match to the attribute value is not possible. With web applications that generate a different ID each time the page is requested, this happens very often. Three types of matching can be used, as follows:

»» element[attribute^='value'] attribute must start with 'value'  
»» element[attribute$='value'] attribute must end with 'value'  
»» element[attribute\*='value'] attribute must contain 'value'

#### Creating Complex Selectors for Unique Locators

The aim here is to create CSS selectors for the elements that refer to work and home information. Use element attributes and partial matches, when possible. Consider the following elements:

<input class="data-input" data-value="phone" type="workPhone">  
<input class="data-input" data-value="address" type="workAddress">  
<input class="data-input" data-value="phone" type="homePhone">  
<input class="data-input" data-value="address" type="homeAddress">

The steps for completion are as follows:

1. Identify the elements that refer to work information; they are the ones that start with work in the type attribute.
2. Construct a CSS selector by combining the element type, the attribute, and the value. For the first element, we would use the following: input, type, and workPhone. The final CSS selector would be as follows: input[type='workPhone'].
3. Repeat steps 1 and 2 for the elements that have workAddress, homePhone, and homeAddress as values in the type attribute.
4. A partial match selector can be used to obtain all of the elements that refer to work information. The selector will look like this: input[type^='work'].
5. After all of the selectors have been found, use them to create the locators with Java.

All major browsers support CSS selectors, due to their native engines. By having a clear understanding of how CSS selectors work and how they can help us, we can build effective locators that are unique and stable.

CSS selectors are the default way to find elements when it is not possible to do so via IDs or class names. They provide a wide range of options that can help us to navigate through the DOM, searching for elements that are not easily reachable. CSS is the better choice when compared to XPath, because all major browsers have native CSS engines; that's why our tests have speedier execution when they use CSS selectors. Learning how to locate elements on a web application is vital in making good use of Selenium WebDriver.

### Activity: Automating Checkout

**Scenario**

Our website includes a checkout to complete the ordering process. We want to make sure that the checkout is working properly for our customers. For now, we will interact with each of the elements on the page, then fill them with data by using Java.

**Aim**

To reinforce what was learned in previous chapters by interacting with elements. You will practice element location through a real-life use case.

**Steps for Completion**

1. Open <https://trainingbypackt.github.io/Beginning-Selenium/lesson_4/activity_4_C-1.html> in Chrome.
2. Open the Chrome DevTools JavaScript to inspect the elements.
3. Locate the first name and last name fields, and write appropriate values on them. Each one has a unique ID.
4. Repeat step 2 for the following fields: email, address, zip, name on card, card number, expiration, and CVV.
5. Locate the drop-down field to select the country, and select one.
6. Locate the checkbox, Save information for next time, and click on it.
7. Locate and select one of the payment methods. Consider that there are no unique IDs for them; a good option would be to use a CSS selector with a partial match, to find a good locator.
8. Locate and click on the Pay button.
9. Collect all of the code pieces that you have built while interacting with the page elements, and combine them in a Java class called CheckoutForm.
10. Run the application by navigating to Run | Run… | CheckoutForm, and observe how your automated script interacts with a checkout form.

*To refer to the detailed steps, go to the Solutions section at the end of this book on Page no. 169.*

## Summary

In this chapter, you learned what the DOM is. You navigated the DOM and created selectors to locate elements in the DOM. You created CSS selectors by using element attributes and partial matches. You learned how to use parent-child and sibling relationships to create selectors for elements that are difficult to reach. We discussed the benefits of using CSS selectors instead of XPath selectors. Finally, you applied what you learned to automated scripts, interacting with proposed web pages.

In the next chapter, we'll work with implicit and explicit waits.

# Waiting for Elements

Nowadays, web applications tend to do a lot of processing in the background or asynchronously before showing elements on the page. As a consequence, web applications do not have a constant speed. This means that we should manage the automation script flow through implicit and explicit waits. In this chapter, you'll learn how to write a stable automation script by waiting for an element to be present.

By the end of this chapter, you will be able to:

* Synchronize an automation script with an implicit wait as well as an explicit wait
* Explain the key differences between implicit and explicit waits
* Synchronize an automation script with a custom written condition

## Implicit Waits

In [*Chapter 4*](https://my.safaribooksonline.com/9781789803815/108d1a2e_e4f5_4d7b_b057_2a37e35d34ac_xhtml)*, Advanced Element Location*, we learned about how to find elements in the **DOM** (**Document Object Model**). But what if one or more elements cannot be found because they have not yet been loaded on to the page? That's when implicit waits are  
helpful.

The TimeOuts interface provides the implicitlyWait method, which is necessary to set an implicit wait. This method receives two parameters (the waiting time and the time unit):

driver.manage().timeouts().implicitlyWait(5, the TimeUnit.  
SECONDS);

Once an implicit wait is set, Selenium will wait up to 5 seconds (according to our preceding example) every time there is a call to the findElement method and the element was not found. During this wait, Selenium will poll to check if the element is already present, and if so, the element will be returned. Note that all of this is done at the server side, so the client has no control at all over the polling intervals; the client will just wait blindly for a response:

driver.findElement(By.id("runTestButton")).click();  
WebElement info = driver.findElement(By.id("info"))

If the element is found, the automation script will continue, and the wait time will be set to zero (0). If the element cannot be found within the established 5 seconds, an exception will be thrown—that is why it is recommended to embed the code of the  
test within a try statement.

The following is an example of a Selenium script with an implicit wait:

public void implicitWaitExample() {  
 WebDriver driver = new ChromeDriver();  
 driver.get("https://trainingbypackt.github.io/Beginning-  
Selenium/lesson\_5/activity\_5\_A-1.html");  
  
 // Set an implicit wait for 5 seconds  
 driver.manage().timeouts().implicitlyWait(5, TimeUnit.  
SECONDS);  
  
 try {  
 /\* Search for a button named runTestButton and click  
on it to start the test\*/  
 driver.findElement(By.id("runTestButton")).click();  
  
 // Verify expected changes to an element affected by  
the test run  
 WebElement info = driver.findElement(By.id("lesson"));  
 if (info.getText().contains("run")) {  
 System.out.println("ImplicitWait worked, the  
element contains 'run'");  
 } else {  
 System.out.println("Something went wrong with  
ImplicitWait, 'run' was not found");  
 }  
 } finally {  
 driver.quit();  
 }  
 }

We are now able to understand what an implicit wait is and how to implement one.

### Creating an Implicit Wait

The aim here is to implement an implicit for an automation script. To complete the automation process, we need to simulate the click of this button and then make our automation script wait before trying to find one of the elements whose attribute values has changed. The steps for completion are as follows:

1. Open <https://trainingbypackt.github.io/Beginning-Selenium/lesson_5/activity_5_A-1.html> on Chrome and open the Dev Tools console.
2. Review and analyze the DOM of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_5/activity_5_A-1.html> file, and see how elements are affected by the functionality of this file.
3. Using IntelliJ IDEA, create a Selenium script that clicks on the available button by using the findElement method. We'll begin with importing the relevant packages and creating the implicitWaitExample() method, as follows:

import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
  
import org.openqa.selenium.chrome.ChromeDriver;  
  
import java.util.concurrent.TimeUnit;  
  
public class Activity5A {  
 public static void main(String[] args) {  
 implicitWaitExample();  
 }

1. Include an implicit wait for any given time. Explore the different options provided by the TimeUnit enumeration, as follows:

private static void implicitWaitExample() {  
 WebDriver driver = new ChromeDriver();  
 driver.get("https://trainingbypackt.github.io/  
Beginning-Selenium/lesson\_5/activity\_5\_A-1.html");  
  
 // Set an implicit wait for 5 seconds  
 driver.manage().timeouts().implicitlyWait(5,  
TimeUnit.SECONDS);

1. Write a findElement command for those elements whose attribute values change once the button is clicked:

try {  
 /\* Search for a button named runTestButton and  
click on it to start the test\*/  
 driver.findElement(By.id("runTestButton")).  
click();

1. Now, check the contents of the element using if … else statements:

if (info.getText().contains("run")) {  
 System.out.println("ImplicitWait worked,  
the element contains 'run'");  
 } else {  
 System.out.println("Something went wrong  
with ImplicitWait, 'run' was not found");  
 }  
 } finally {  
 driver.quit();  
 }  
 }  
}

1. Compile and run the script.

## Explicit Waits

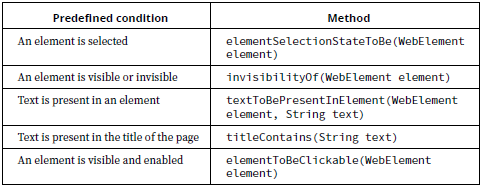
Implicit waits work under the assumption that the element(s) we are searching for might take some time to appear on the DOM. Hence, an implicit wait will wait for every element that is part of our automation script.

On the other hand, explicit waits allow us to control the automation script flow for one or more specific elements, and under predefined conditions. The WebDriverWait and ExpectedConditions classes are used for setting explicit waits. The following lines set a 5-second explicit wait until the title of the page contains the string "Explicit". Note that if the title is updated before the 5-second waiting time, the automation script will continue to the next step:

WebDriverWait wait = new WebDriverWait(driver, 5);  
wait.until(ExpectedConditions.titleContains("Explicit"));

It is important to highlight that explicit waits act only on the elements involved in the wait process. Any other element won't be affected, and it is expected to be present when Selenium interacts with it. As a side note, explicit waits are also known as active waits.

Here are some of the pre-defined conditions supported by the ExpectedConditionsclass (<http://seleniumhq.github.io/selenium/docs/api/java/index.html>):



The following is an example of a Selenium test with an explicit wait:

public void explicitWaitExample() {  
 WebDriver driver = new ChromeDriver();  
 driver.get("https://trainingbypackt.github.io/Beginning-  
Selenium/lesson\_5/activity\_5\_B-1.html");  
  
 try {  
 /\* Search for a button named runTestButton and click on it  
to start the test\*/  
 driver.findElement(By.id("runTestButton")).click();  
  
 // Set an explicit wait  
 WebDriverWait wait = new WebDriverWait(driver, 5);  
 wait.until(ExpectedConditions.titleContains("Explicit"));  
  
 // Verify expected changes  
 if (driver.getTitle().startsWith("Explicit")) {  
 System.out.println("ExplicitWait worked, the element  
contains 'Explicit'");  
 } else {  
 System.out.println("Something went wrong with  
ExplicitWait, 'Explicit' was not found");  
 }  
 } finally {  
 driver.quit();  
 }  
}

We are now able to understand what an explicit wait is and how to implement one.

### Activity: Creating an Explicit Wait

**Before you Begin**

* Open <https://trainingbypackt.github.io/Beginning-Selenium/lesson_5/activity_5_B-1.html> on Chrome and switch to source view.
* Use IntelliJ IDEA for the creation of a Selenium test.

**Scenario**

The web page we are performing the test on contains a button. By clicking on it, the values of the attributes of some elements are changed. To complete the automation process, we need to simulate the click of this button and then make our test wait before trying to find one of the elements whose attributes' values have changed.

**Aim**

To create a Selenium test that clicks on a button, and implement an explicit wait for the test.

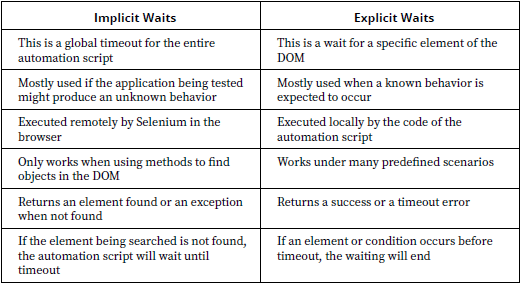
**Steps for Completion**

1. Review and analyze the DOM of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_5/activity_5_B-1.html> file, and see how elements are affected by the functionality of this file.
2. Create a Selenium test that clicks on the available button by using the findElementmethod.
3. Choose any of the elements whose attribute values are changed after clicking on the test button. Create an explicit wait for the chosen object.
4. For each one of the instructions in step 3, check the contents of the title by using the if … else statement.
5. Compile and run the test.

*To refer to the detailed steps, go to the Solutions section at the end of this book on page no. 171.*

## Implicit Versus Explicit Waits

Now that we have learned what implicit and explicit waits are, let's review how they compare to each other:



Some of the items in the preceding table allow us to determine that explicit waits are, most of the time, the better choice for our test.

On one hand, the capacities of implicit waits are limited due to the fact that they only work when using methods to find objects in the DOM. Explicit waits are more flexible and versatile because they allow for the setting of many test scenarios based on a range of possible pre-conditions.

On the other hand, implicit waits are global timeouts for the entire automation script. Hence, every time the script encounters a findElement command, it will poll until the element is found. But if the element is not found, the script will keep waiting until the timeout. These conditions will affect the performance and increase the execution time of the script.

We are now able to understand the differences between implicit and explicit waits.

## Waiting for an Element with a Custom Written Condition

As we saw earlier, explicit waits allow you to build a test using many pre-conditions that are provided by the ExpectedConditions class. But what if these conditions are not sufficient to recreate a test scenario? The good news is that it is possible to implement the ExpectedCondition interface to create custom wait conditions as needed.

Custom conditions can be used in different scenarios, such as:

* When waiting for an element to be found
* When waiting for an element's attribute value to be updated
* When waiting for an element to be visible or invisible
* When waiting for DOM events

### Creating Custom Waits for Finding Elements

The following code sets a custom wait of five seconds while attempting to find an element with ID "info". Once it finds the element, the automation script will return that same element:

WebElement element = new WebDriverWait(driver, 5)  
 .until(new ExpectedCondition<WebElement>() {  
 public WebElement apply(WebDriver d) {  
 return d.findElement(By.id("lesson"));  
 }  
 });

#### Creating a Custom Wait (Waiting for an Element)

The aim here is to implement a custom wait (waiting for an element). Consider the following web page at <https://github.com/TrainingByPackt/Beginning-Selenium/blob/master/docs/lesson_5/exercise_5_1.html>:

<!doctype html>  
<html lang="en">  
 <head>  
 <meta charset="utf-8">  
 <title>Waiting for an element</title>  
 </head>  
  
 <body>  
 <div class="pricing-header px-3 py-3 pt-md-5 pb-md-4 mx-auto  
text-center">  
 <h1 class="display-4" id="lesson">Lesson 5, Exercise 1  
(Pricing)</h1>  
<h5 class="lead" id="instruction">Click on the button to  
start a custom wait test</h5>  
 </div>  
  
 <div class="container">  
 <div class="card-deck mb-3 text-center">  
 <div class="card mb-4 box-shadow">  
 <div class="card-header">  
 <h4 class="my-0 font-weight-normal">Waiting for: </h4>  
//[…]

The steps for completion are as follows:

1. Review and analyze the DOM of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_5/exercise_5_1.html> file, and see how elements are affected by the functionality of this file.
2. Create a Selenium test that clicks on the available button:

driver.findElement(By.id("runTestButton")).click();

1. Choose any of the elements whose attribute values are changed after clicking on the test button. Create a custom wait related to the chosen element:

WebDriverWait wait = new WebDriverWait(driver, 5);  
 wait.until(ExpectedConditions.  
titleContains("Explicit"));

1. For each one of the instructions in *step 3*, check the contents of the element's texts (as we did in the previous activity):

if (driver.getTitle().startsWith("Explicit")) {  
  
 System.out.println("ExplicitWait worked,  
the element contains 'Explicit'");  
  
 } else {  
  
 System.out.println("Something went wrong  
with ExplicitWait, 'Explicit' was not found");  
 }

1. Compile and run the test.
2. You may repeat steps *3* and *4* for different objects.

### Creating Custom Waits so That an Element's Attribute Values Can Be Updated

When running a test, the attribute of one or more elements can be changed at runtime. For instance, the text of a label can be modified with a new message, or a button can be enabled once certain steps have been completed. In such cases, a custom wait can be written on the attribute value of any of those elements.

This custom wait will wait 5 seconds before checking if the "disabled" attribute of the runTestButton element has changed to "false":

Boolean active = new WebDriverWait(driver, 5)  
 .until(new ExpectedCondition<Boolean>() {  
 public Boolean apply(WebDriver d) {  
 return d.findElement(By.id("runTestButton")).  
getAttribute("disabled").contains("false");  
}  
});

#### Creating a Custom Wait (Waiting for an Element's Attribute Updates)

Based on the explanation of this concept, create a test with a custom wait for this web page. The aim is to implement a custom wait, which waits for an element's attribute value to be updated.

<!doctype html>  
<html lang="en">  
 <head>  
 <title>Waiting for an element attribute value</title>  
 </head>  
  
 <body>  
 <div class="pricing-header px-3 py-3 pt-md-5 pb-md-4 mx-auto  
text-center">  
 <h1 class="display-4" id="lesson">Lesson 5, Exercise 2  
(Pricing)</h1>  
<h5 class="lead" id="instruction">Click on the button to  
start a custom wait test</h5>  
 </div>  
  
 <div class="container">  
 <div class="card-deck mb-3 text-center">  
 <div class="card mb-4 box-shadow">  
 <div class="card-header">  
 <h4 class="my-0 font-weight-normal">Waiting for: </h4>  
//[…]

The steps for completion are as follows:

1. Review and analyze the DOM of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_5/exercise_5_2.html> file, and see how elements are affected by the functionality of this file.
2. Create a Selenium test that clicks on the button that's available:

driver.findElement(By.id("runTestButton")).click();

1. Choose any of the elements whose attributes values are changed after clicking on the test button. Create a custom wait related to the value of an attribute of the chosen element:

WebDriverWait wait = new WebDriverWait(driver, 5);  
  
 wait.until(ExpectedConditions.  
titleContains("Explicit"));

1. For each one of the instructions in *step 3*, check the contents of the element's texts (as we did in the previous activities).

if (driver.getTitle().startsWith("Explicit")) {  
  
 System.out.println("ExplicitWait worked,  
the element contains 'Explicit'");  
  
 } else {  
  
 System.out.println("Something went wrong  
with ExplicitWait, 'Explicit' was not found");  
 }

1. Compile and run the test.
2. You may repeat steps 3 and 4 for different objects.

### Creating Custom Waits for an Element's Visibility

Not only can the values of attributes of an element be modified during runtime. A button, for example, can also be set to disappear after certain steps have been completed. In such cases, a custom wait can be written based on the visibility of the elements.

This custom wait will wait 5 seconds before checking as we did the "runTestButton" element is visible or not:

Boolean active = new WebDriverWait(driver, 5)  
 .until(new ExpectedCondition<Boolean>() {  
 public Boolean apply(WebDriver d) {  
 return d.findElement(By.id("runTestButton")).  
isDisplayed();  
}  
});

#### Creating a Custom Wait (Waiting for an Element's Visibility)

Based on the explanation of this concept, we'll create a test with a custom wait for this web page. The aim is to implement a custom wait (waiting for an element's visibility) for this web page. Consider the following web page at <https://github.com/TrainingByPackt/Beginning-Selenium/blob/master/docs/lesson_5/exercise_5_3.html>:

<!doctype html>  
<html lang="en">  
 <head>  
 <meta charset="utf-8">  
 <title>Waiting for an element visibility</title>  
 </head>  
  
 <body>  
 <div class="pricing-header px-3 py-3 pt-md-5 pb-md-4 mx-auto  
text-center">  
 <h1 class="display-4" id="lesson">Lesson 5, Exercise 3  
(Pricing)</h1>  
<h5 class="lead" id="instruction">Click on the button to  
start a custom wait test</h5>  
 </div>  
  
 <div class="container">  
 <div class="card-deck mb-3 text-center">  
 <div class="card mb-4 box-shadow">  
 <div class="card-header">  
 <h4 class="my-0 font-weight-normal">Waiting for: </h4>  
//[…]

The steps for completion are as follows:

1. Review and analyze the DOM of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_5/exercise_5_3.html> file, and see how elements are affected by the functionality of this file.
2. Create a Selenium test that clicks on the button that's available:

driver.findElement(By.id("runTestButton")).click();

1. Choose an element whose visibility changes after clicking on the test button. Create a custom wait related to that element:

WebDriverWait wait = new WebDriverWait(driver, 5);  
  
 wait.until(ExpectedConditions.  
titleContains("Explicit"));

1. Check the contents of the element's texts (as we did in the previous activities).

if (driver.getTitle().startsWith("Explicit")) {  
  
 System.out.println("ExplicitWait worked,  
the element contains 'Explicit'");  
  
 } else {  
  
 System.out.println("Something went wrong  
with ExplicitWait, 'Explicit' was not found");  
 }

1. Compile and run the test.

## Summary

In this chapter, we looked at how to control the automation script flow by using waits. We then created implicit and explicit waits. We saw the differences between these waits. Finally, we created custom waits by implementing the ExpectedConditions class.

In the next chapter, we'll explore the Page Object Model and model test automation scripts.

# Page Object Model

In the last chapter, you learned how to write automation scripts that allow you to control the various load times of elements on a web page. Now that you know how to use Selenium and a test framework, you can easily create hundreds of unstructured tests. However, in an application, automation code is as important as production code, and the appropriate patterns and standards should be used.

The Page Object Model is a well-known pattern that introduces a middle layer to tests, which can help to reduce code duplication and make the code easier to understand. In the Page Object Model, low-level implementation details are hidden.

In this chapter, the you will learn how to model tests that better reflect the tested web application.

By the end of this chapter, you will be able to:

* Explain the principles behind the Page Object Model
* Model a test automation script by implementing the Page Object Model
* Implement nested page object instances

## Introduction to the Page Object Model

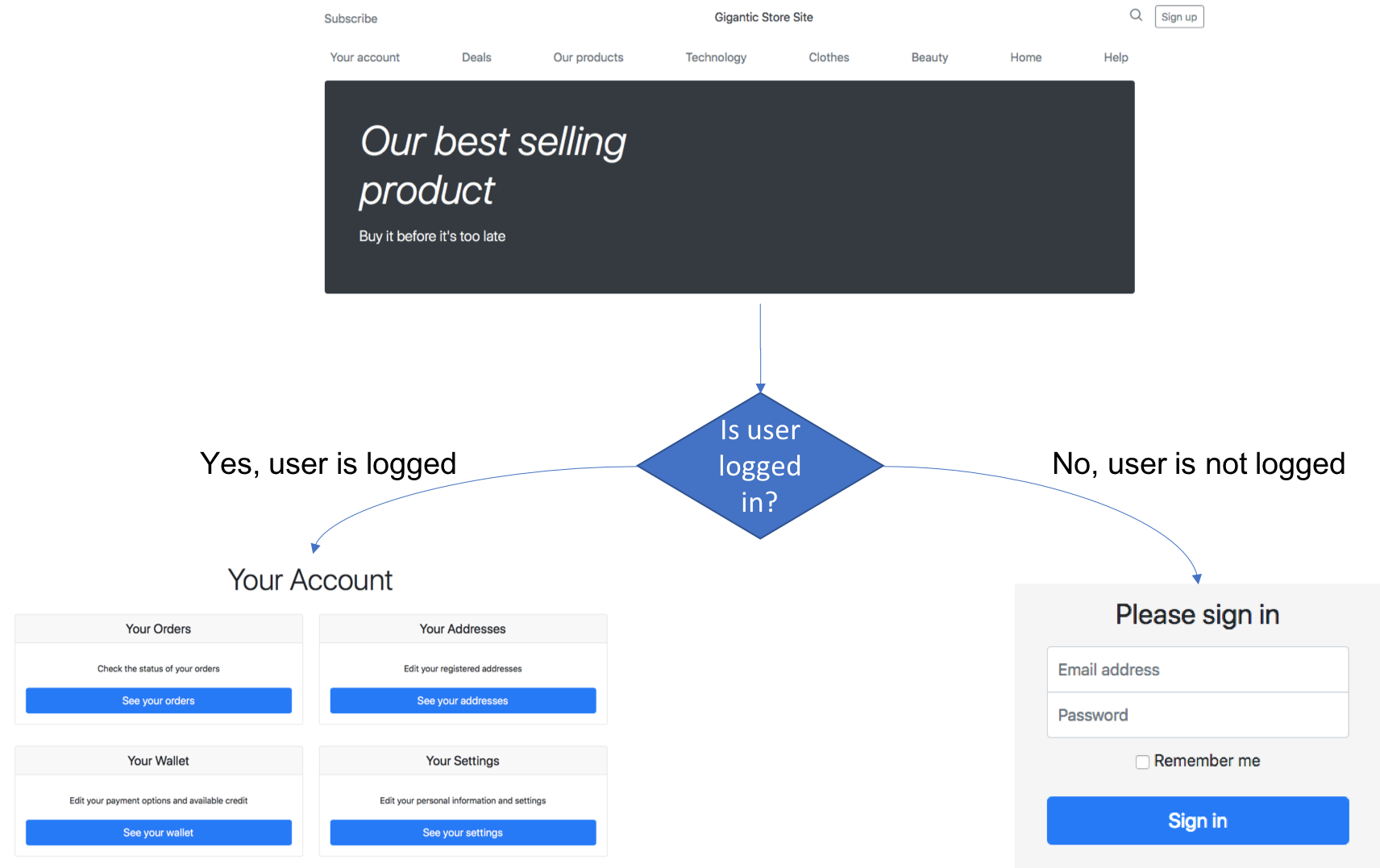
Most of our previous chapters included practical sub-sections. Although they were based on different testing scenarios, they shared one aspect: they involved writing a single automation script, to test a single web page. In real life, however, we will find different circumstances, as follows:

Web applications consist of many web pages that interact with one another.

Even if we only have a single-page web application, we might be interested in writing and executing more than one test on it.

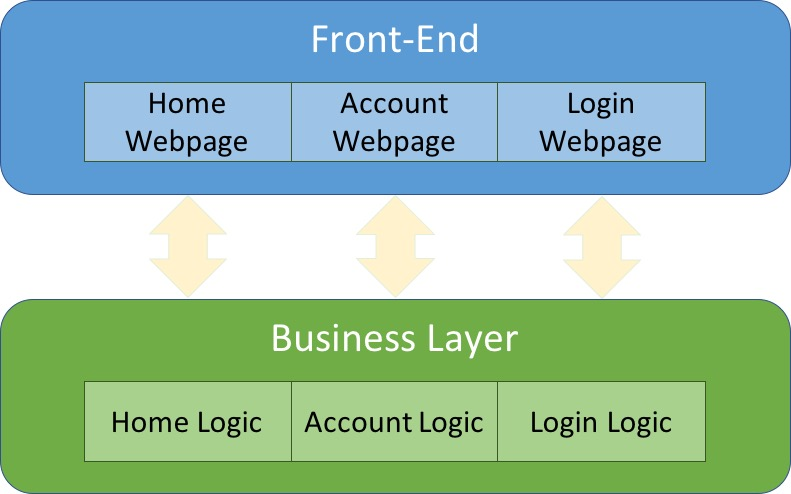
### Understanding the Architecture of Web Applications

Let's take a look at the following diagram of the application of an online store, including three web pages:



If a user clicks on the Your Account option on the Home Webpage, the application will check whether they have already logged in. If they have, it will redirect them to their Account Webpage. If they have not, it will redirect them to the Login Webpage.

Here, we have three web pages that interact with each other; this interaction depends on the user's input. The development team behind this application most likely built a model similar to the following one (for simplification purposes, our example includes only two layers):



A few aspects of the preceding diagram are important to highlight, as follows:

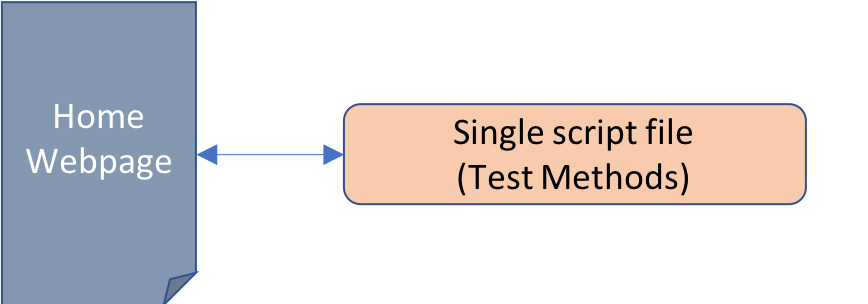
* The **Front-End Layer** presents forms that the user can see and interact with.
* The **Business Layer** executes and validates the business rules of each web page.
* For each web page, there is a corresponding component in the Business Layer.

The reason behind this development approach is maintainability; we can describe this as follows:

* If the business rules regarding the login process change (implementing a new algorithm to encrypt the password, for example), the developers will only have to make changes in the **Login Logic** element, keeping the **Login Webpage**unaltered.
* The opposite is true; in this case: if a new layout or presentation form is required for the **Login Webpage**, changes can be made without having to modify the **Login Logic** component.

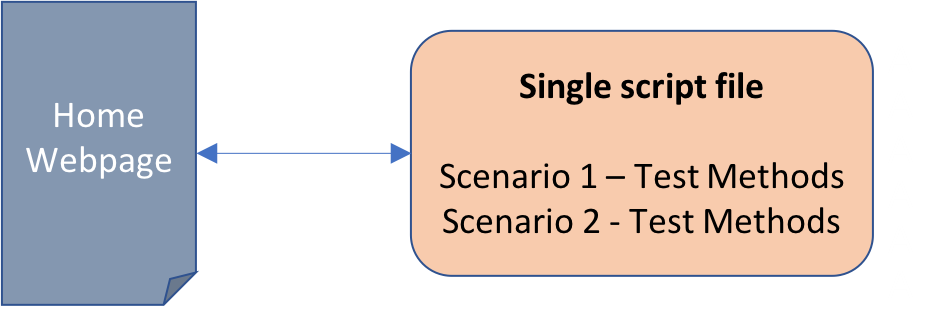
### Applying Web Application Architecture to Test Automation Scripts

Throughout this book, we have dealt with single-page applications, for which we wrote single-file scripts. This approach is the simplest, as illustrated by the following diagram; we write one script (in a single file) for each web page of our application:



This method works fine for a while. But what if we encounter a new scenario for testing the same web page? We have two options. We can continue to use the same script file that we used previously, or we can create a new script file for the new test scenario. Let's examine these two options, as follows.

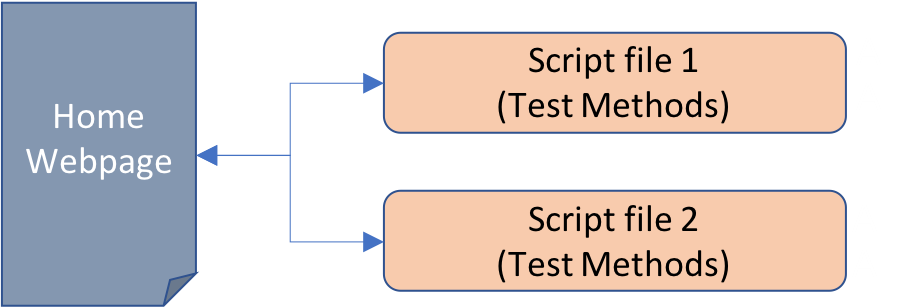
**Option 1**: We can add the code for the new test to the old script file. A home page requires two test scenarios, and we can write them both in the same script file, as shown in the following diagram:



However, if we choose the preceding option, the following applies:

* We risk breaking the previous test.
* Maintaining the code will become more difficult, as more lines of code are added.

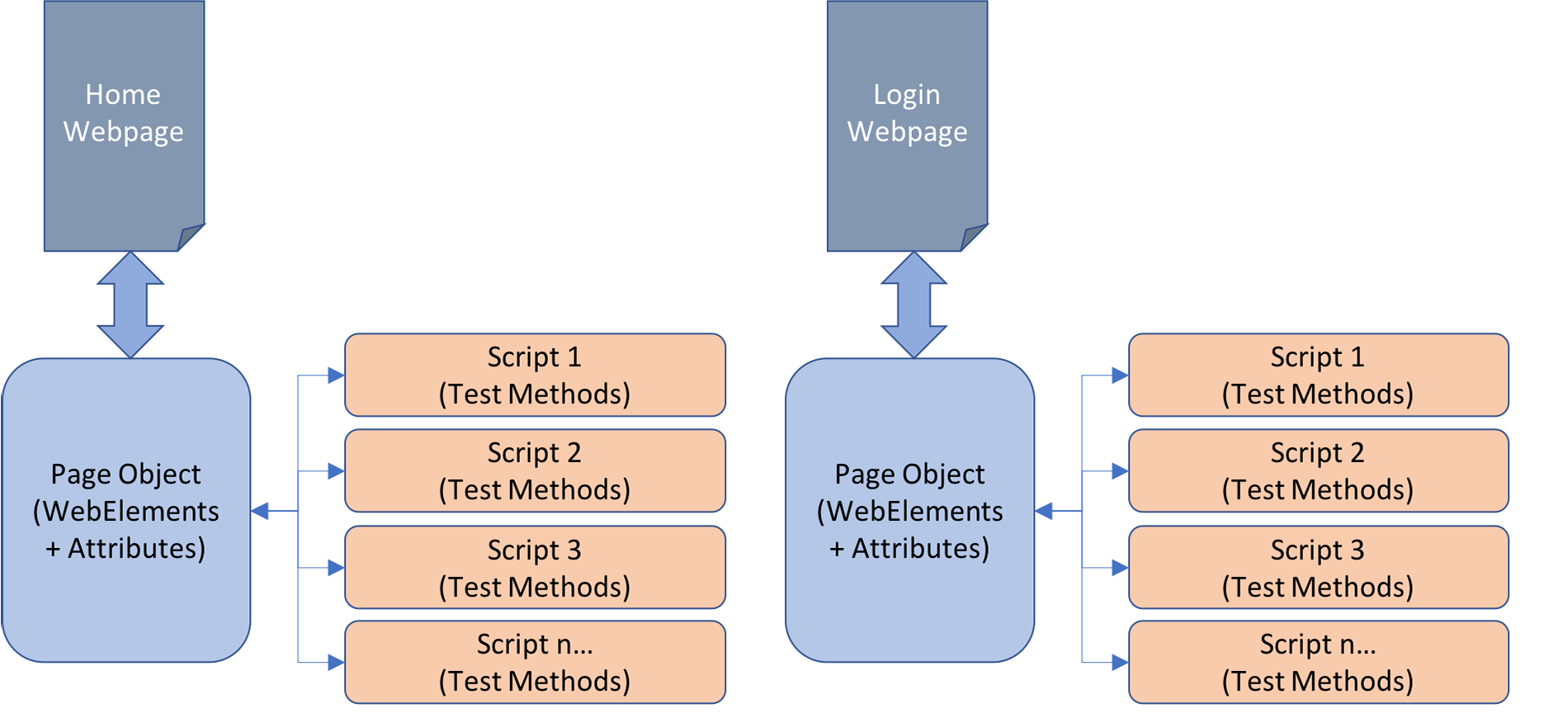
**Option 2**: We can create the new test in a new script file. A home page requires two test scenarios, and we can write each one in a separate script file, as follows:



However, if we choose the preceding option, the following applies:

* We will be replicating the code related to the navigation of the web page(locating and interacting with the web elements of the page, for example).
* If changes are made to the web elements of the page in the future, we will have to update both the old and new scripts.
* Suppose that instead of two automation scripts, you have a hundred scripts for a single web page—things will get complicated.

Neither of the preceding options are optimal. The best practice is to utilize an approach similar to the multi-tier architecture in test automation scripts (even for a single-page web application). By applying software development best practices, we can separate the navigation and validation tasks for automation script writing. We can do so through the **Page Object Model** (**POM**), as illustrated in the following diagram:



The preceding diagram shows us what the POM consists of and how it changes the way that we design and write automation scripts:

* There is a **Page Object** class for each web page in the application. The Page Object class includes the code specific to handling the page's web elements, including finding its elements, accessing its attributes, and more.
* There are as many automation scripts as there are scenarios that we might need to test for each web page. Every script will instantiate the page object that is testing and will only handle the methods of the test.

Advantages of POM

Implementing POM into automation script writing can provide the following benefits  
to a testing project:

* It separates WebElements navigation code and automation script code, which provides for cleaner and easier-to-understand code.
* It creates a repository of **Page Objects**, which allows them to be reused for different purposes and testing scenarios.
* If the design or layout of WebElements and/or their attributes are changed, we will only have to modify its corresponding Page Object, not every script that uses it.

You should now understand what the Page Object Model is and how it improves the automation script-writing process.

## Modeling a Web Application Using Page Objects

As we stated previously, a Page Object is a representation of a web page that accesses and interacts with the elements of the web page and their attributes. We model a web application as follows:

* By creating Page Objects for each web page that we would like to test. The set of Page Objects for our project will constitute our Object Repository.
* By writing automation scripts that make use of those Page Objects.

### Creating a Page Object

A Page Object is an abstract representation of a web page that includes its elements and its attributes; through a Page Object, we can perform automation scripts that follow the POM approach.

The following steps describe how to create a Page Object:

1. Create a new Java file by using your IDE of choice.
2. Define a package for your Object Repository and include it at the top of every Page Object file that you create, as follows:

package com.beginningselenium.examples.pageobjects;

1. Since we are creating a Page Object for the **Age Calculator** application, it might be a good idea to name our class AgeCalculatorPage, as shown here:

public class AgeCalculatorPage {  
}

1. Our AgeCalculatorPage class must include all WebElements of the web page that we might use in the automation script. We can include them as follows (it is a good practice to use the same names or IDs that are given to the names in the web page):

private WebElement dayOfBirth;  
private WebElement age;

1. Set variables and methods to handle the driver, as follows:

private WebDriver driver;  
private String url = "https://trainingbypackt.github.io/  
Beginning-Selenium/lesson\_6/exercise\_6\_1.html";  
  
public void open() {  
 this.driver.get(url);  
 }  
 public void close() {  
 this.driver.quit();  
 }

1. Write a constructor to the AgeCalculatorPage class that maps the elements of the page to the variables of the class, as follows:

public AgeCalculatorPage(WebDriver webDriver) {  
 driver = webDriver;  
 }

1. Write operations to read the values of the required WebElements:

public String getAge() {  
 age = driver.findElement(By.id("age"));  
 return age.getAttribute("value");  
 }

1. Our Age Calculator application includes a button that calculates the age and the zodiac sign of the user. We can also model this by using Page Object Model:

public void calculate(String day, String month, String  
year) {  
 getDayOfBirth().sendKeys(day);  
 getMonthOfBirth().sendKeys(month);  
 getYearOfBirth().sendKeys(year);  
 getCalculate().click();  
 }

You should now understand how to implement a Page Object for a web page.

#### Creating a Page Object For Age Calculator

The aim here is to create a Page Object for the Age Calculator application. Consider the Age Calculator application. The Age Calculator is a single-page application that takes the user's date of birth and calculates their age and zodiac sign:

<html lang="en">  
<head>  
 <title>Lesson 6 - Age Calculator</title>  
</head>  
  
<body class="bg-light">  
 <div class="container">  
 <div class="py-5 text-center">  
 <h2>Lesson 6 - Age Calculator</h2>  
 </div>  
  
 <div class="row">  
 <div class="col-md-4 order-md-2 mb-4">  
 <h4 class="d-flex justify-content-between align-  
items-center mb-3">  
 <span class="text-muted">Your data</span>  
 </h4>  
 <ul class="list-group mb-3">  
 <li class="list-group-item d-flex justify-  
content-between lh-condensed">  
 <div>  
 <h6 class="my-0">Age</h6>  
 </div>  
 <span class="text-muted" id="age"></span>  
 </li>  
 <li class="list-group-item d-flex justify-content-  
between lh-condensed">  
 <div>  
 <h6 class="my-0">Zodiac Sign</h6>  
 </div>  
 <span class="text-muted"  
id="zodiacSign"></span>  
 </li>  
 </ul>  
 </div>  
 <div class="col-md-8 order-md-1">  
 <h4 class="mb-3">Enter your information</h4>  
 <form>  
 <div class="row">  
//[…]

*For the complete code, visit*[*https://bit.ly/2NVcpGu*](https://bit.ly/2NVcpGu)*.*

The steps of achieving our aim are as follows:

1. Review and analyze the DOM of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_6/exercise_6_1.html> file.
2. Create a Page Object that recreates the WebElements of the Age Calculator application. Follow the best practices while naming classes, elements, and methods, and make sure to include the following:
   * The name of the package
   * The libraries required for the Page Object to work
   * A class constructor method

Firstly, import all the packages:

package com.beginningselenium.examples.pageobjects;  
  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;

1. Include all of the WebElements of the web page that you might have to use in the automation script:

public class AgeCalculatorPage {  
 //WebElements  
 private WebElement dayOfBirth;  
 private WebElement monthOfBirth;  
 private WebElement yearOfBirth;  
 private WebElement age;  
 private WebElement zodiacSign;  
 private WebElement calculate;

1. Create the class constructor, ensuring that you include all of the WebElements that have to be initialized:

///WebDriver  
 private WebDriver driver;  
 private String url = "https://trainingbypackt.  
github.io/Beginning-Selenium/lesson\_6/exercise\_6\_1.  
html";  
  
//Class Constructor  
public AgeCalculatorPage(WebDriver webDriver) {  
 driver = webDriver;  
 }

1. Include methods to read the values of the required elements:

//Methods to read values from required WebElements  
 public String getAge() {  
 age = driver.findElement(By.id("age"));  
 return age.getText();  
 }  
  
 public String getZodiacSign() {  
 zodiacSign = driver.findElement(By.  
id("zodiacSign"));  
 return zodiacSign.getText();  
 }  
 public WebElement getDayOfBirth() {  
 dayOfBirth = driver.findElement(By.  
id("dayOfBirth"));  
 return dayOfBirth;  
 }  
 public WebElement getMonthOfBirth() {  
 monthOfBirth = driver.findElement(By.  
id("monthOfBirth"));  
 return monthOfBirth;  
 }  
 public WebElement getYearOfBirth() {  
 yearOfBirth = driver.findElement(By.  
id("yearOfBirth"));  
 return yearOfBirth;  
 }  
 public WebElement getCalculate() {  
 calculate = driver.findElement(By.  
id("calculate"));  
 return calculate;  
 }

1. Create a method that opens and closes the WebDriver, and a method that recreates clicking on the Calculate button:

//Methods to open and close the WebDriver  
 public void open() {  
 this.driver.get(url);  
 }  
 public void close() {  
 this.driver.quit();  
 }  
  
 //Method to execute the test  
 public void calculate(String day, String month,  
String year) {  
  
 getDayOfBirth().sendKeys(day);  
 getMonthOfBirth().sendKeys(month);  
 getYearOfBirth().sendKeys(year);  
 getCalculate().click();  
 }

1. Compile the class.

### Creating an Automation Script

In the previous section, we created the AgeCalculatorPage Page Object that exposes the elements of the web page for the AgeCalculator application. For each test scenario that requires the use of AgeCalculatorPage, we can create an automation script in a different Java file. From each script, we can access the web page's elements, through the AgeCalculatorPage Page Object. When doing so, we should keep the following in mind:

1. We should define a package for all of our tests and include it at the top of every script that we create, as follows:

package com.beginningselenium.examples.scripts;

1. We should import the package that our object repository belongs to (and all other required libraries), as follows:

import com.beginningselenium.examples.pageobjects.  
AgeCalculatorPage;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.chrome.ChromeDriver;

1. Since we are creating a script for the AgeCalculator web page, it would be a good practice to name our class AgeCalculatorScript:

public class AgeCalculatorScript {  
}

1. We should always create a method where the script's code can be placed:

public checkAgeCalculator() {  
}

1. To use the Page Object, we must create an instance of the AgeCalculatorPage class, and then start the driver:

WebDriver driver = new ChromeDriver();  
AgeCalculatorPage ageCalculatorPage = new  
AgeCalculatorPage(driver);  
 ageCalculatorPage.open();

1. We can execute the test by using the calculate method of the AgeCalculatorPage:

ageCalculatorPage.calculate("11", "February", "1982");

1. We should verify that the values of age and zodiac sign provided by the application correspond to the expected values. In our example, we mentioned February 11, 1982, which means that the user is 36 years old (the current year, as of writing this chapter, is 2018) and belongs to the Aquarius zodiac sign:

if (ageCalculatorPage.getAge().equals("36")) {  
System.out.println("Age was calculated correctly!");  
} else {  
System.out.println("There was an error in the age  
calculation");  
}  
  
if (ageCalculatorPage.getZodiacSign().equals("Aquarius")) {  
System.out.println("Zodiac sign was calculated  
correctly!");  
} else {  
System.out.println("There was an error in the zodiac sign  
calculation");  
}

1. We can stop the test as follows:

ageCalculatorPage.close();

You should now understand how to implement an automation script with the Page Object Model.

#### Creating an Automation Script for Age Calculator

The aim here is to create an automation script that uses the AgeCalculator Page Object that was created in the previous section. Consider the Age Calculator application. The Age Calculator is a single-page application that takes the user's date of birth and calculates their age and zodiac sign:

<html lang="en">  
<head>  
 <title>Lesson 6 - Age Calculator</title>  
</head>  
  
<body class="bg-light">  
 <div class="container">  
 <div class="py-5 text-center">  
 <h2>Lesson 6 - Age Calculator</h2>  
 </div>  
  
 <div class="row">  
 <div class="col-md-4 order-md-2 mb-4">  
 <h4 class="d-flex justify-content-between align-  
items-center mb-3">  
 <span class="text-muted">Your data</span>  
 </h4>  
 <ul class="list-group mb-3">  
 <li class="list-group-item d-flex justify-  
content-between lh-condensed">  
 <div>  
 <h6 class="my-0">Age</h6>  
 </div>  
 <span class="text-muted" id="age"></span>  
 </li>  
 <li class="list-group-item d-flex justify-content-  
between lh-condensed">  
 <div>  
 <h6 class="my-0">Zodiac Sign</h6>  
 </div>  
 <span class="text-muted"  
id="zodiacSign"></span>  
 </li>  
 </ul>  
 </div>  
 <div class="col-md-8 order-md-1">  
 <h4 class="mb-3">Enter your information</h4>  
 <form>  
 <div class="row">  
//[…]

The steps of achieving our aim are as follows:

1. Review and analyze the DOM of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_6/exercise_6_1.html> file.
2. Review the Page Object of the Age Calculator application, as defined in the <https://github.com/TrainingByPackt/Beginning-Selenium/blob/master/docs/lesson_6/lesson6/src/main/java/com/beginningselenium/examples/pageobjects/AgeCalculatorPage.java> file.
3. Create a new Java file for the automation script using IntelliJ IDEA. Make sure to include the following:
   * The name of the package.
   * The libraries required for the Page Object to work.
   * A class method to encapsulate the code of the script.

package com.beginningselenium.examples.scripts;  
  
import com.beginningselenium.examples.pageobjects.  
AgeCalculatorPage;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.chrome.ChromeDriver;  
public class AgeCalculatorScript {  
  
 public static void main(String[] args) {  
 checkAgeCalculator();  
 }

1. Using the Calculate method of the AgeCalculatorPage, provide input values to execute the test. Make sure to include the expected results, to compare them with the results provided by the application.

private static void checkAgeCalculator() {  
WebDriver driver = new ChromeDriver();  
// Create an instance of AgeCalculatorPage class  
and open it  
AgeCalculatorPage ageCalculatorPage = new  
AgeCalculatorPage(driver);  
ageCalculatorPage.open();  
// Start the test by means of the calculate  
method  
ageCalculatorPage.calculate("11", "February",  
"1982");  
// Verify results  
if (ageCalculatorPage.getAge().equals("36")) {  
System.out.println("Age was calculated  
correctly!");  
} else {  
System.out.println("There was an error in  
the age calculation");  
}  
if (ageCalculatorPage.getZodiacSign().  
equals("Aquarius")) {  
System.out.println("Zodiac sign was  
calculated correctly!");  
} else {  
System.out.println("There was an error in  
the zodiac sign calculation");  
}  
ageCalculatorPage.close();  
}  
}

1. Compile and run the script.

### Activity: Implementing the POM on a Multi-Page Application

**Before You Begin**

1. Open Chrome. Go to <https://trainingbypackt.github.io/Beginning-Selenium/lesson_6/activity_6_B-1.html> and open the DevTools console.
2. Use IntelliJ IDEA to create a Selenium script.

**Scenario**

The Gigantic Store provides the layout for a wholesale store. It includes a menu, which contains different options. Clicking on any of the options will redirect the user to a different web page.

In this activity, we will create a Page Object for the home page, and for any other page of the application. Finally, we will create an automation script that makes use of the Page Objects.

**Aim**

To implement an automation script with the Page Object Model on a multi-page web application.

**Steps for Completion**

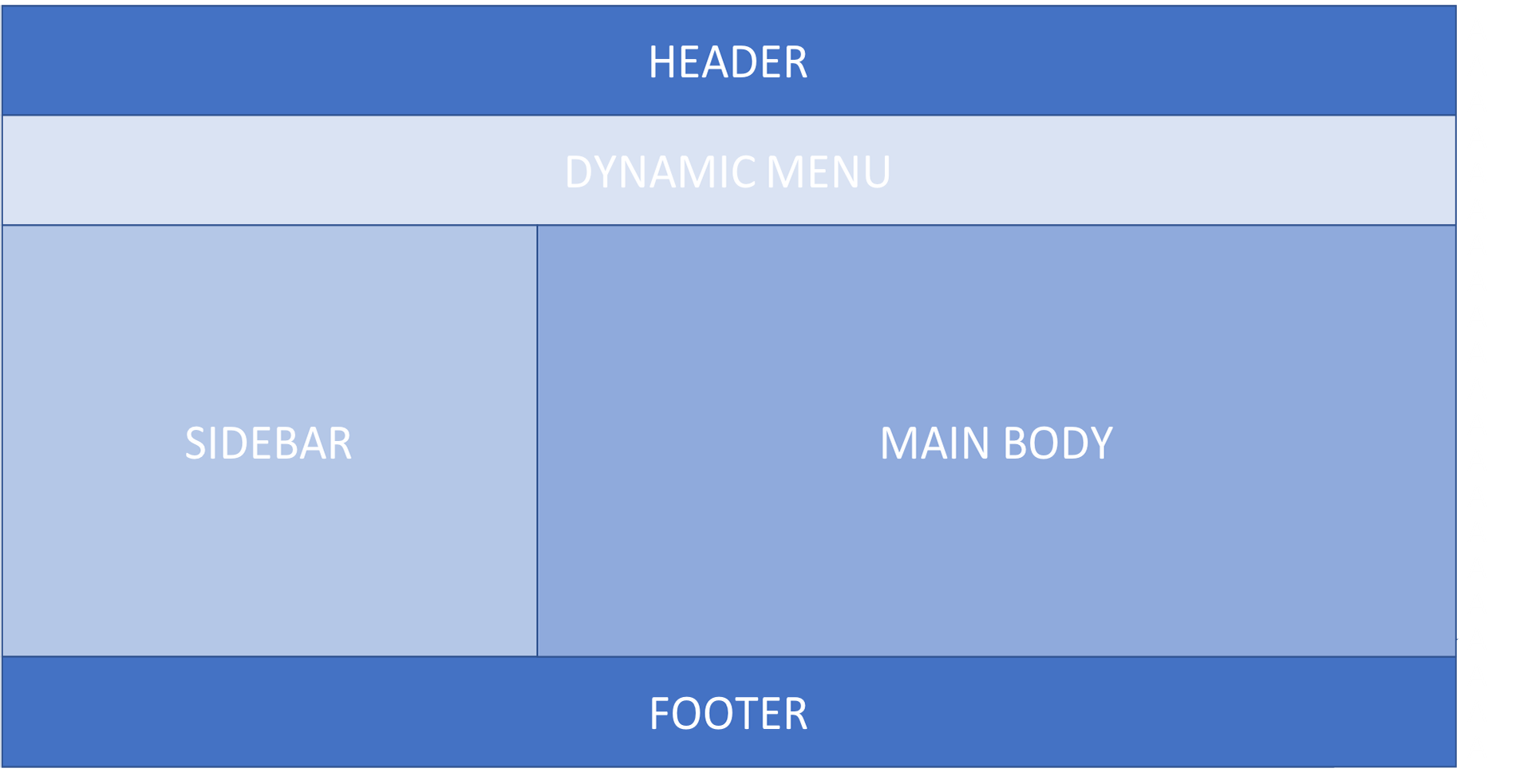
1. Create a Page Object for the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_6/activity_6_B-1.html> file (which is considered the home page of the Gigantic Store application):
   1. Create a new Java file by using your IDE of choice.
   2. Define a package for our Object Repository and include it at the top of every Page Object file that you create.
   3. Include all of the libraries required, such as WebDriver, ChromeDriver, and WebElement.
   4. Since we are creating a Page Object for the Home page, it would be a good practice to name our classHomePage.
   5. Set variables and methods to handle the driver. Use the open and close functions.
   6. Write a constructor to the HomePage class.
   7. Write an operation to click on any of the options of the menu.
   8. Create an operation that uses the clickOption from any other page of the application. For this activity, we will choose the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_6/activity_6_B-1/deals.html> page.
2. Create a Page Object for the page that was chosen in *step 1*, as follows:
   1. Create a new Java file by using your IDE of choice.
   2. Include this Page Object in the same Object Repository that includes the HomePage class.
   3. Include all of the required libraries, such as WebDriver, ChromeDriver, and WebElement.
   4. Since we are creating a Page Object for the Deals page, it would be good practice to name our class DealsPage.
   5. Set variables and methods to handle the driver.
   6. Set the variables of the required web elements. The Deals page includes an element with the ID quote, with specific text.
   7. Write a constructor to the DealsPage class that verifies that the Deals page has been loaded and initializes the quote element.
   8. Write an operation to read the value of the quote element.
3. Create an automation script for the HomePage class. The test should include clicking on the Deals option in the menu and verifying that the **Deals** page loads:
   1. Define a package for the test.
   2. Import the Object Repository of the Page Objects, and all other required libraries, such as DealsPage, HomePage, WebDriver, and ChromeDriver.
   3. Since we are creating a script for the Age Calculator application, it would be good practice to name our class HomeScript.
   4. Create a method where the script's code can be placed. Name it checkHomeAndDealsPage.
   5. To use the Page Object, we must create an instance of the HomePage class, and then start the driver.
   6. Start the test by invoking the clickDeals method of the HomePage class. This method returns a DealsPage class object.
   7. Verify the test.
   8. Stop the test.
4. Compile and run the test.

*To refer to the detailed steps, go to the Solutions section at the end of this book on Page no. 172.*

## Implementing Nested Page Object Instances

Pose these questions to the class. The aim is to discuss the complexity of web application structures and web page layouts:

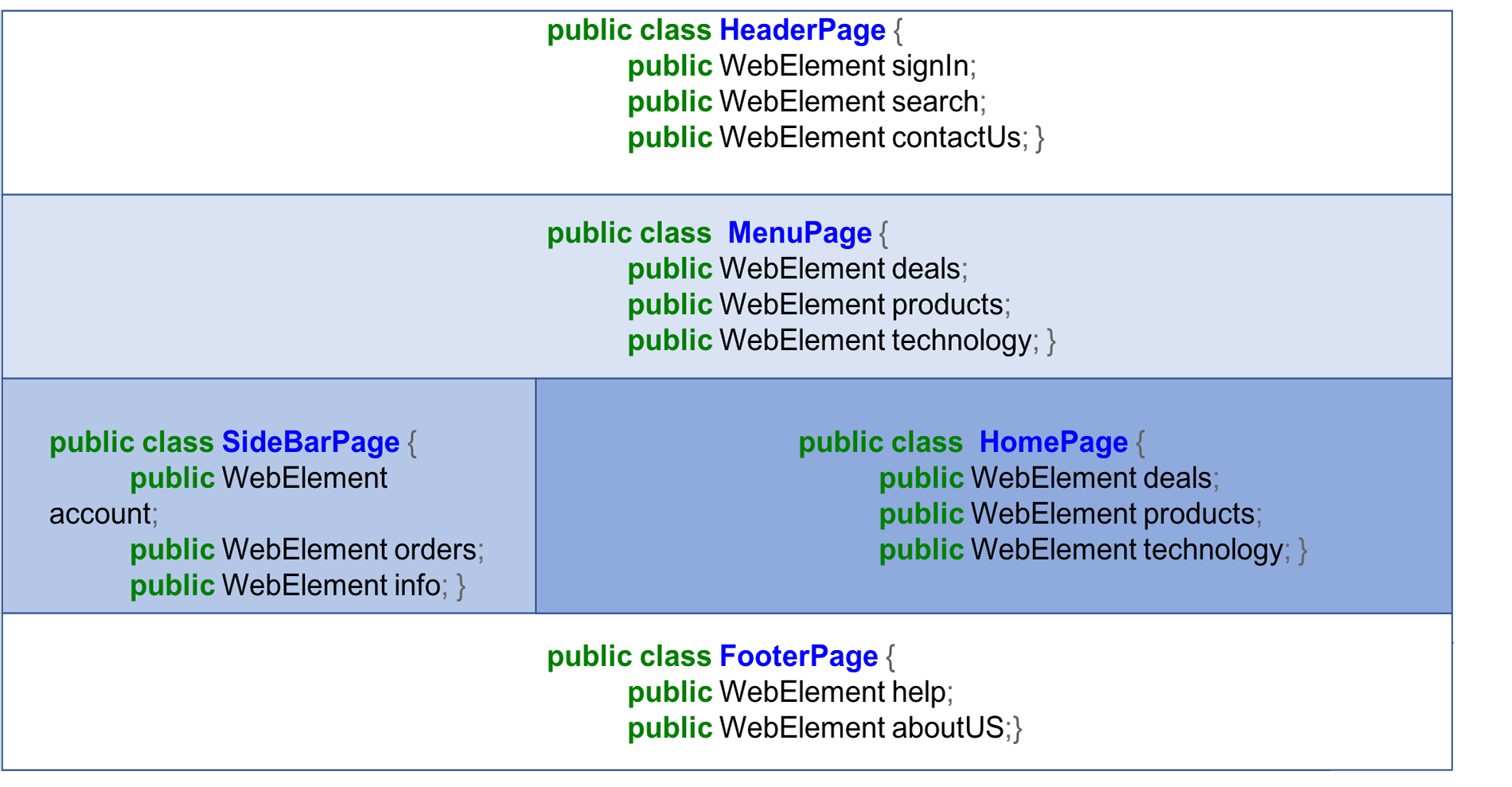
In *Modeling a Web Application Using Page Objects*, of the current chapter, you learned how to implement the Page Object Model for a web application with one or more web pages. However, applications sometimes involve more complex structures. For example, the home page of an application can consist of many different sections, as illustrated by the following diagram:



Each of these sections can include several elements, with a wide variety of attributes. Although it is possible to write the corresponding Page Object (HomePage class) for a web page like this, it would end up having so many attributes, methods, and lines of code, that the whole purpose of using the Page Object Model—maintainability—would be lost.

A different approach to the POM involves implementing nested Page Object instances. We could model our web page as follows:

* We could create a Page Object for each section of the home web page. Each of these Page Objects would include, and handle, its own web elements and attributes.
* We could create a Page Object for the home web page. This Page Object would have the Page Objects that we created for each section of the website as its attributes. In other words, this Page Object would be a wrapper to access the previously created Page Objects. Its attributes would be variables of the Page Objects in the sections, listed as follows:



By modeling the home page sections as Page Objects, we increase the number of classes, but improve the maintainability and reusability. The nested Page Object approach can also be utilized to model pieces of functionality that do not belong to a single page and are available across several pages in the application (such as the sign-in button or a search field).

### Activity: Implementing Nested Page Object Instances

**Before You Begin**

Open Chrome and go to <https://trainingbypackt.github.io/Beginning-Selenium/lesson_6/activity_6_B-1.html>.

**Scenario**

The Gigantic Store application home page contains three sections: a header, a navigation menu, and a banner that displays information, according to the option clicked by the user. We will model this home page by employing the nested Page Object approach, and we will write an automation script that makes use of the nested Page Objects.

**Aim**

To implement nested Page Object instances in a web application with different pages, such as a home page and a products page.

**Steps for Completion**

1. Analyze the DOM and layout of the <https://trainingbypackt.github.io/Beginning-Selenium/lesson_6/activity_6_B-1.html> file.
2. Create a Page Object for the Header section.
3. Create a Page Object for the Menu section. Import packages such as WebDriver, By, and WebElement.
4. Create a Page Object for the Banner section. Include methods to read the information on the banner, such as getMainInfo and getSecondaryInfo.
5. Create a Page Object for HomePage.
6. Create a Page Object for the Products page. Include methods to navigate the header and read the banner.
7. Write an automation script that uses the Page Objects that were created. The script should include methods to verify the results in the products page and the home page.
8. Compile and run the script.

*To refer to the detailed steps, go to the Solutions section at the end of this book on Page no. 176.*

## Summary

In this chapter, you learned about the importance of utilizing the Page Object Model to optimize and improve test automation maintainability. You created a Page Object. You created an automation script by implementing the Page Object Model. Finally, you created an automation script by implementing nested Page Object instances.

In the next chapter, we'll do more than just write scripts—we'll write tests.

# Writing Tests

For the last six chapters, we have been writing Selenium automation scripts. But these are not actual tests, just scripts that use WebDriver to control a browser and execute commands against a given web page. Selenium helps us automate browsers, but we want to do more than that. We want to validate if the web application is working as expected. To do this, we need to add a test framework in to the mix. By combining Selenium and a test framework, we can write tests to check and assert a web application's elements and behavior.

By the end of this chapter, you will be able to:

* Explain what a test framework is and how to choose one
* Create test scripts and suites
* Validate and view results

## What is a Test Framework?

## Choosing a Test Framework

## Creating Test Scripts and Suites

### Annotations

### The TestNG.xml File

### Activity: Creating a TestNG Test

## Validating and Viewing Results

### Activity: Viewing the Results

## Summary

# Analysis and Troubleshooting

## Analyzing a Test Report

### General Recommendations

#### Analyzing a Test Result

## Tracking Down Timing Errors

### How to Deal with Potential Timing Errors

#### Tracking Down Timing Errors with Synchronization Points

## Separating Real Issues from Flaky Tests

### Identifying Flaky Tests

### Decreasing the Chances of Flaky Tests

## Summary

# Using a Selenium Grid

## Configuring and Connecting to a Local Grid

### Launching the Grid Hub

#### Verifying That the Grid Hub Is Running

### Adding Selenium Nodes to the Grid Hub

#### Verifying That a Selenium Instance Has Been Registered

### Activity: Running a Script Against a Grid Hub

## Configuring and Connecting to a Network Grid

### Adding Selenium Nodes Instances Running on Different Machines to the Grid Hub

### Activity: Creating a Small Selenium Grid with a Remote Node

## Connecting to a Third-Party Service

### Activity: Running a Test on a Third-party Service

## Summary

# Solutions

## Chapter 1: Getting Started

### Activity: Automation Brainstorming

### Activity: Selenium Planning

## Chapter 2: WebDriver Functionality

### Activity: Starting and Finalizing a Script

### Activity: Resizing and Moving Windows with a Selenium Automation Script

## Chapter 3: WebElement Functionality

### Activity: Filling in a Form and Submitting It

### Activity: Locating Elements

## Chapter 4: Advanced Element Location

### Activity: Automating Checkout

## Chapter 5: Waiting for Elements

### Activity: Creating an Explicit Wait

## Chapter 6: Page Object Model

### Activity: Implementing the POM on a Multi-Page Application

### Activity: Implementing Nested Page Object Instances

## Chapter 7: Writing Tests

### Activity: Creating a TestNG Test

## Chapter 9: Using a Selenium Grid

### Activity: Running a Script Against a Grid Hub

### Activity: Creating a Small Selenium Grid with a Remote Node

### Activity: Running a Test on a Third-Party Service

# Other Books You May Enjoy