

Selenium WebDriver

From Foundations To Framework



Yujun Liang & Alex Collins

Table of Contents

Introduction	1.1
Part 1: Fundamentals	1.2
Chapter 1: First steps	1.3
Chapter 2: Locating elements on a page	1.4
Chapter 3: Interacting with elements on a page	1.5
Chapter 4: Examining a page	1.6
Chapter 4: Making maintainable tests using the Page Object pattern	1.7
Chapter 6: What to do when something goes wrong	1.8
Part 2: WebDriver APIs in depth	1.9
Chapter 7: Managing WebDriver	1.10
Chapter 8: Windows, pop-ups, and frames	1.11
Chapter 9: Unicorns and other beasts: exotic features of web pages	1.12
Chapter 10: Executing JavaScript using the JavascriptExecutor interface	1.13
Chapter 11: What you need to know about different browsers	1.14
Chapter 12: Wrapping WebDriver and WebElement	1.15
Part 3: Page based automation framework	1.16
Chapter 13: Forming a framework	1.17
Chapter 14: Encapsulating and grouping elements	1.18
Chapter 15: Automating a page flow	1.19
Chapter 16: Examining HTML tables	1.20
Chapter 17: Automating jQuery datepicker	1.21
Chapter 18: Datepicker framework	1.22
Appendices	1.23
Appendix A: Selenium grid	1.23.1

Introduction

This book is a hands-on guide to dozens of specific ways you can use to get the most of WebDriver in your test automation development. This practical handbook gives you instantly-useful solutions for important areas like interacting with and testing web applications and using the WebDriver APIs. As you read, you'll graduate from WebDriver fundamentals to must-have practices ranging from how to interact with, control and verify web pages and exception handling, to more complex interactions like page objects, alerts, and JavaScript, as well as, mobile testing, and much more. Finally, you'll learn how to build your own framework. By the end of the book, you'll be confident and skilled at testing your web applications with WebDriver.

About the technology

Web applications are difficult to test because so much depends on the way a user interacts with individual pages. The Selenium WebDriver web testing framework helps you build reliable and maintainable test automation for your web applications across multiple browsers, operating systems and programming languages. Much like a human, it can click on links, fill out forms, and read the web pages, and unlike a human, it does not get bored. WebDriver can do nearly anything you ask it to—the trick is to come up with a unified approach to testing. Fortunately, that's where this book really shines.

What's inside

- Specific, practical WebDriver techniques
- Interacting with, controlling, and testing web applications
- Using the WebDriver APIs
- Making maintainable tests
- Automated testing techniques

Testimonials

Quotes from our early access reviewers:

Excellent coverage of a key technology in the web testing space.

An essential book for anyone interested in doing WebDriver integration testing. You should have some familiarity with Java development (including basic use of Maven). It starts with basic Selenium WebDriver usage but there's plenty more. It's clear that the authors have been using this technology in a professional setting for quite some time as the book is littered with one technique after another which can be used to address problems one can expect when testing real world web applications.

This book is a very practical guide to Selenium WebDriver. The book is loaded with practical examples with their solutions. I have already used techniques to solve problems at work.

It's a really good introduction to the framework and I like the way the authors have attempted to provide practical solutions to the problems one faces when trying to automate certain types of tests.

Acknowledgements

Thank you to the following contributors:

- entropicrune
- Edko24
- PulwerJR

Errata and Discussion

If you find any errors or problems with this book, or if you want to talk about the content:

<https://github.com/selenium.webdriver-book/manuscript/issues>

About the reader

This book assumes you're comfortable reading code in Java or a similar language and that you know the basics of building and testing applications. No WebDriver experience is required.

About the authors



Yujun Liang is a Technical Agile Coach who teaches agile software development technologies including test automation using Selenium WebDriver. He used to work for ThoughtWorks and helped clients build automation testing for web applications with rich user interaction and complex business logic.



Alex Collins is a Technical Architect in the UK, a technology blogger, public speaker, and OSS contributor. Alex has been working with Selenium WebDriver since 2011.

Copyright © 2016 Yujun Liang and Alex Collins

Part 1: Fundamentals

In this section we will introduce you to Selenium WebDriver. We'll teach common techniques that are useful in writing tests, such as locating, interacting and verifying elements. We'll also show you how to make your code more maintainable using Page Objects and how to deal with errors*. You'll be able to write code for many common web pages by the end of it.

Chapter 1: First Steps

This chapter covers

- What is WebDriver?
- Why choose WebDriver?
- "Hello WebDriver!"

Nowadays, more and more business transactions are carried out on the Internet through web pages built by people. Some websites are simple enough that they can be set up by one or two people, but some websites are so complex that they are built by hundreds or even thousands of developers. Before each release, the site must be tested to make sure it is free of critical bugs. It is time-consuming to test the whole site manually, and as the site grows, so does the cost of testing. More than that, as time passes, a new feature that was well-tested when it first became available may be forgotten about later—we risk of a loss of consistency and quality, and as a result bugs in what we thought were solid pieces of functionality creep in.

In the textile industry, manual labor dominated the process of making clothes for a long time. When weaving machines were invented, productivity improved dramatically.

The same thing is happening in software testing. Just as weaving machines changed the textile industry, we are now building "automatic testing machines" to replace manual testing, to improve the productivity, quality, and consistency of the software.

Since its inception in 2008, **Selenium WebDriver** (also known as Selenium 2) has established itself as the de facto web automation library.

Before Selenium WebDriver, there was Selenium 1.0, which enabled automation by injecting JavaScript into web pages. WebDriver is a re-invention of that idea, but is more reliable, more powerful, and more scalable.

Selenium has evolved, and so has the World Wide Web. **HTML5** and **CSS3** are now standard; AJAX rich web applications are no longer even cutting edge. This means that web automation is now a more complex and interesting topic.

This chapter will rapidly cover the basics, making sure that by the end of it you understand the basic architecture can write basic code.

In this chapter we'll introduce WebDriver, what it is, how it works, and reasons for choosing it. We'll also briefly talk about some of the tools we used in this book, the ones we'd recommend to all developers.

What is WebDriver?

Selenium WebDriver automates web browsers. It sits in the place of the person using a web browser. Like a user, it can open a website, click links, fill in forms, and navigate around. It can also examine the page, looking at elements on it and making choices based on what it sees.

The most common use case for WebDriver is automated testing. Until recently, to run a regression test on your website, you'd need to have a set of scripts that would have to be manually executed by developers or QAs. Any reports would need to be manually collated too. This can be both time-consuming and costly. Instead, WebDriver can be used to execute those scripts, and automatically gather reports on how successful they were, at the push of a button. Each subsequent execution will be no more expensive than the first.

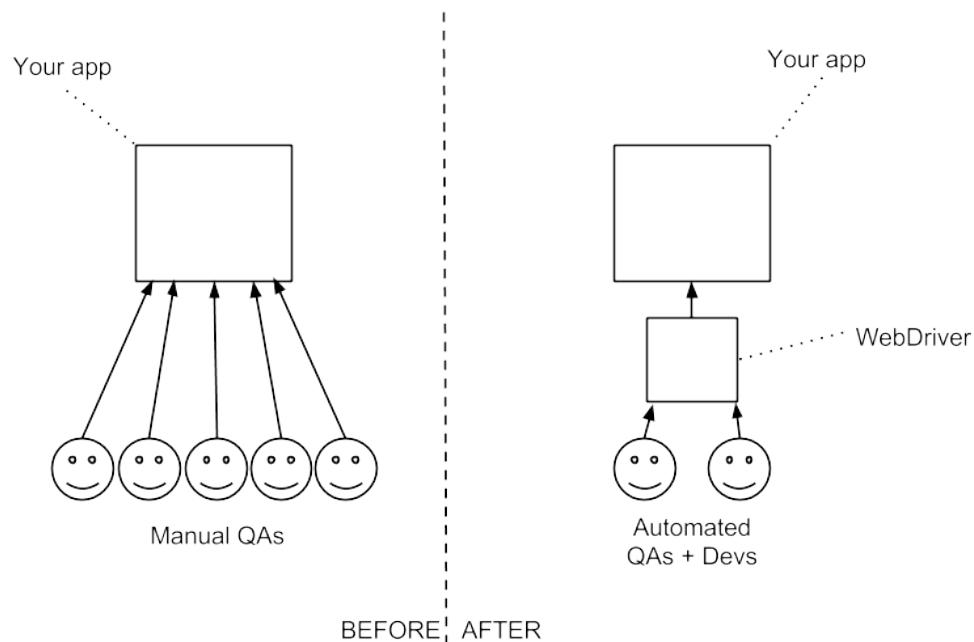


Figure 1. Before WebDriver

Long gone are the days when you needed to create one version on your website for the pervasive and notoriously standards non-compliant Internet Explorer 6, and another for other browsers. While most modern browsers are much more consistent in their behavior, the way a web page looks or acts can still greatly vary as the number of different browsers, operating system, and platforms in common use has increased. You can still have a high-value customer complain that they can't access your site. Historically, the only way to mitigate this was to have an army of QAs manually test on a variety of different configurations, a time-consuming and costly process. WebDriver can run tests on different operating systems and different browser configurations, and in a fraction of the time of a human being. Not only that, you can use it to run them much more consistently and reliably than a manual tester.

Applications and websites provide useful services, but sometimes these are only accessible by web pages. Another use case for WebDriver is to make those pages accessible to applications via WebDriver. You might have an administration application written several years ago and a client or Product Owner has asked for some actions on it to be automated. But maybe no one knows where the source code is. It might be much easier to use WebDriver to automate this task.

How WebDriver works

WebDriver works in all major browsers and with all major programming languages.

How is this possible? Well, WebDriver has several interacting components:

1. A **web browser**.
2. A **plugin or extension** to the browser that lives inside the browser, which itself contains a server that implements the WebDriver JSON API.
3. A **language binding** (in our case Java) that makes HTTP requests to that API.

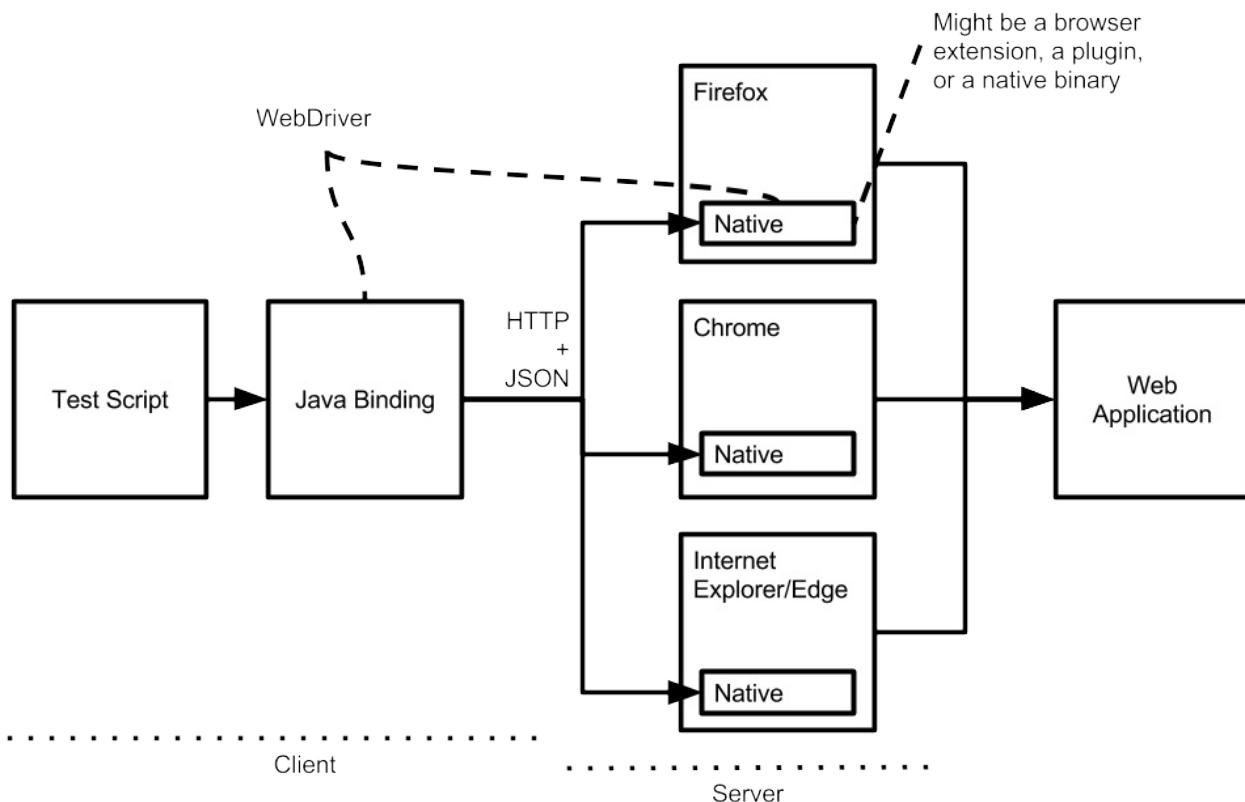


Figure 2. Web driver diagram

When you start code that uses WebDriver, it will open up the browser, which in turn starts the plugin. You can then send requests to perform the actions you want, such as clicking on links or typing text. As a plugin only needs to implement the JSON API, people have written plugins for all major browsers. To use a browser that has a plugin, you just need to implement a client to the JSON protocol.

This means that all the major browsers and all the major programming languages support WebDriver.

The plugin can usually be seen in the browser's preferences, such as in figure 1.3.

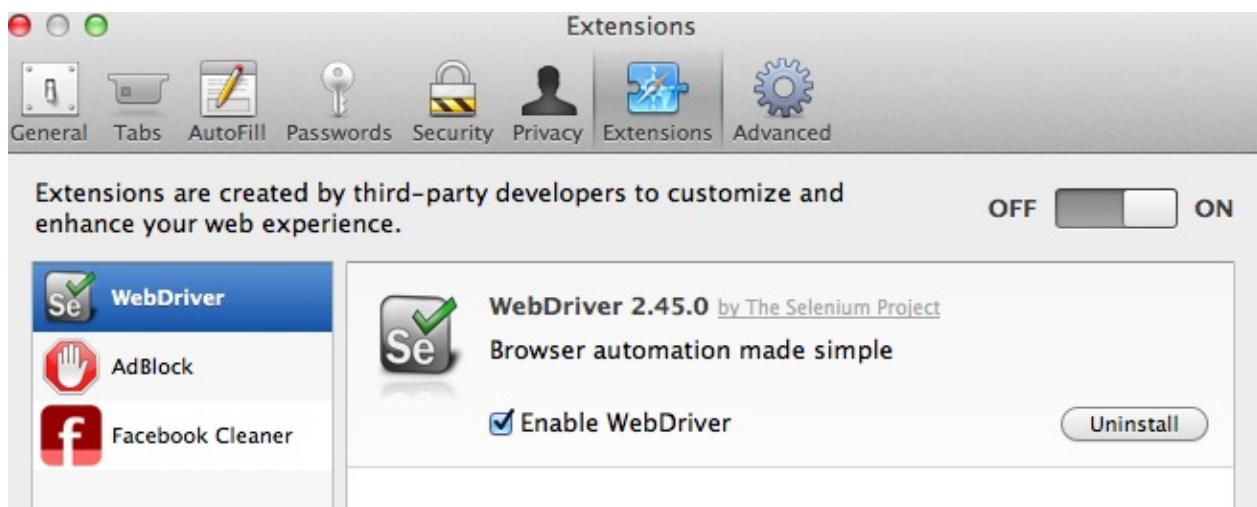


Figure 3. Safari Extensions panel

Why choose WebDriver?

There are a number of great reasons to choose WebDriver:

- WebDriver can run browsers **locally and remotely** with minimal configuration changes.
- WebDriver is **supported by major browser vendors**: both Firefox and Chrome are active participants in WebDriver's development.
- WebDriver more closely mimics a user. Where possible it uses **native events** to operate, to make it accurate and stable.
- WebDriver is **Open Source Software (OSS)**. This means that it is both free and is supported by an excellent community.
- WebDriver supports all major operating systems such as **OS X, Windows, and Linux**. It also has support for **Android** and **iOS**.
- WebDriver is going to become a **W3C standard**, so you can expect that it will be supported for a long time.
- WebDriver doesn't suffer from some of the problems that Selenium 1.0 suffered from, such as with uploading files, or handling pop-ups.
- WebDriver has a more **concise syntax** than Selenium 1.0, making it faster to write code.

What WebDriver cannot do

WebDriver provides a way to control a web browser, but that is all. When you buy a new car, you get a manual that will tell you how to operate the radio and how to change the oil. But that manual won't tell you the best place to get your car serviced, or teach you how to drive. Like driving a car, there are things you must do for yourself. Here are some things WebDriver does not do:

- WebDriver doesn't have the control of the timing of the elements appearing on the web page. Some might appear later and you'll need to handle this yourself.
- WebDriver does not know when things have changed on the page, so you can't ask it to tell you when things have changed.
- WebDriver doesn't provide many utilities for writing your code. You need to write these yourself.
- WebDriver doesn't provide built-in support for page elements that are composed of multiple elements, such as JavaScript calendars.
- WebDriver does not provide a framework to write your code in. JUnit is a natural choice.
- WebDriver doesn't manage the browser. For example, you need to clean up after you have used it.
- WebDriver won't install or maintain your browsers. You need to do this yourself.

We'll cover all these important tasks in this book.

The history of Selenium

Selenium is a suite of web testing tools, including **Selenium IDE**, **Selenium RC**, **Selenium WebDriver**, and **Selenium Grid**. The earliest Selenium is called **Selenium Core**, which came out of ThoughtWorks's Chicago office developed by **Jason Huggins**. It was written to simulate a human user's action with Internet Explorer. It was different from Record/Replay tool types from other vendors, since it didn't require an additional GUI tool to support its operation. It just needed Java, which most developers had already installed on their machines.

Later, **Shinya Kasatani** developed a Firefox plugin called **Selenium IDE** on top of Selenium Core. Selenium IDE is a graphic interface allowing users to record a browser navigation session, which can be replayed afterwards. Selenium IDE integrated with Firefox and provided the same Record/Replay function as the other proprietary tools.

Since Selenium IDE is a free tool, it soon captured a big market share among QAs and business analysts who didn't have the necessary programming skills to use Selenium Core. Later Selenium Core evolved into Selenium RC ("RC" meaning "Remote Control"), along with Selenium Grid, which allowed tests can be run on many machines at the same time.

Selenium WebDriver was originally created by Simon Stewart at Thoughtworks. It was originally presented at Google Test Automation Conference, and this can still be seen online <https://www.youtube.com/watch?v=tGu1ud7hk5I>.

In 2008, Selenium incorporated WebDriver API and formed Selenium 2.0. Selenium WebDriver became the most popular choice among the Selenium tool suite, since it offers standardized operation to various Browsers through a common interface, WebDriver. In favor of this new simplified WebDriver API, Selenium RC has been deprecated and its usage is no longer encouraged. The developers who maintain Selenium also provided a migration guide helping Selenium RC users migrating from Selenium RC to WebDriver.

Today, when people talk about "Selenium," they're usually talking about Selenium WebDriver.

Why it is called Selenium?

Jason Huggins joked about a competitor named Mercury in an email, saying that you can cure mercury poisoning by taking selenium supplements. That's where the name Selenium came from.

The tools you need to get started

On top of a computer, access to the Internet, and a development environment, you will need some additional pieces of software to get started.

Why we use Java in this book

While we're aware that many developers won't be using Java as their main language. We chose it as the language for this book because we wanted to write for the most people possible, and Java is the most popular programming language.

The API to WebDriver is similar in all languages. The languages of the web – JavaScript, CSS, and HTML – are the same regardless of the language you're writing your tests in. If you're using one of the many languages that WebDriver supports, such

as C#, Ruby or Python, you should be able to replicate many of the techniques in this book.

Java Development Kit (JDK)

As Java is among the most popular and widely used development languages, we will be using it throughout this book. Java 8 introduces a number of features, such as streams and lambda expressions, that make it faster and more efficient to work with WebDriver.

You can check to see if (and which version of) the JDK is already installed from a terminal using the `javac` command:

```
$ javac -version  
javac 1.8.0_20
```

Note that it is typical to refer to a Java version by the middle digit of the full version number, so "Java 1.8" is usually known as "Java 8."

Linux users can install Java using the `yum` or `apt` package managers. Windows and OS X users can download it from Oracle at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

Apache Maven

Throughout this book we will use the Apache Maven build tool for managing our code. The main reason for this is because Maven can manage the many dependencies that we need to create an application. You can check to see if you have Maven installed from the terminal:

```
$ mvn -version  
Apache Maven 3.3.1
```

If you do not have it installed, Linux users can install it using their package manager (e.g. Apt or Yum), OS X users can install it using the Homebrew package manager (<http://brew.sh>)

For example (on OS-X using Brew):

```
brew install maven
```

Or (on Ubuntu Linux):

```
sudo apt-get install maven
```

Windows users can download it from the Apache Maven website at <https://maven.apache.org>.

Google Chrome

Chrome browser is the best supported browser. Unlike other browsers, it is available on every platform, it's standards compliant, and has the simple out-of-the-box integration with WebDriver.

As usual, Linux users can install Chrome using their package manager; otherwise you can download Chrome from Chrome.

Later on the book we will look at other browsers such as Firefox, but having Chrome installed now will get you through the first few chapters.

Git

You'll need to install Git if you want to check out the source code for this book. On OS-X using Brew:

```
brew install git
```

Or on Ubuntu Linux using Apt:

```
sudo apt-get install git
```

If you use Windows, you can download it from <https://git-scm.com/> .

The test project

As part of this project, we have put all the source code into the Git version control system. This contains all the sample code, as well as a small website the code runs against.

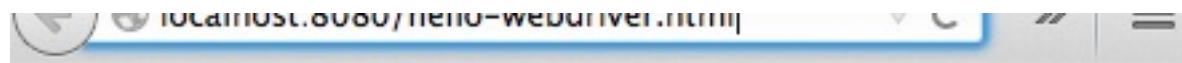
You can get this by running these commands:

```
git clone https://github.com/selenium.webdriver-book/source.git  
cd source
```

The project has a built-in web server that can be started by entering the following:

```
mvn jetty:run
```

You can view the website it creates at <http://localhost:8080/hello-webdriver.html>. You should see a page similar to figure [Hello WebDriver](#). This forms the basis of many of the tests in the project, so you'll probably want to keep it running all the time.



Hello WebDriver!

Welcome to the first example!

Figure 4. Hello WebDriver

When you are done, press `Ctrl+C` to quit the server.

If you want to find examples from the book in the code, look for the package named after the chapter. For example, if you're looking for chapter one's examples, then they can be found in `src/test/java/swb/ch01intro`.

To run all the tests with the book, run the following:

```
mvn verify
```

Instructions on how to run with different browsers can be found in the `README.md` file.

“Hello WebDriver!”

Let's look at an example of using WebDriver to automate a basic task, and end up with a working example. A WebDriver automation script usually consists of several operations:

1. Create a new WebDriver, backed by either a local or remote browser.

2. Open a web page.
3. Interact with that page, for example clicking links or entering text.
4. Check whether the page changes as expected.
5. Instruct the WebDriver to quit.

Create a directory with this `pom.xml`:

`pom.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>hello-webdriver</groupId>
  <artifactId>hello-webdriver</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-chrome-driver</artifactId> (1)
      <version>LATEST</version> (2)
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <source>1.8</source> (3)
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
```

```
<artifactId>maven-failsafe-plugin</artifactId>
<version>2.18.1</version>
<executions>
    <execution>
        <goals>
            <goal>integration-test</goal> (4)
            <goal>verify</goal>
        </goals>
    </execution>
</executions>
</plugin>
</plugins>
</build>

</project>
```

1. You can choose a different browser, e.g. `selenium-firefox-driver` is for the Firefox browser.
2. Always use the latest version that is available.
3. Compile using the latest version of Java – Java 1.8.
4. Make sure that tests are run using Maven’s failsafe plugin.

To start the driver, you’ll need a special binary program to start it up. For Chrome, this is called `chromedriver` and can be found at

<https://sites.google.com/a/chromium.org/chromedriver/downloads>. Download it and then save it into the root of the project.

Create `src/test/java/swb/intro/HelloWebDriverIT.java` :

[HelloWebDriverIT.java](#)

```
package swb.ch01intro;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By; (1)
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

import static org.junit.Assert.assertEquals;

public class HelloWebDriverIT { (2)

    private WebDriver driver;

    @Before
    public void setUp() throws Exception {
        System.setProperty("webdriver.chrome.driver", "chromedriver"); (3)
        driver = new ChromeDriver(); (4)
    }

    @After
    public void tearDown() throws Exception {
        driver.quit(); (5)
    }

    @Test
    public void helloWebDriver() throws Exception {

        driver.get("http://localhost:8080/hello-webdriver.html"); (6)

        WebElement pageHeading
            = driver.findElement(By.tagName("h1")); (7)

        assertEquals("Hello WebDriver!",
                    pageHeading.getText()); (8)
    }
}
```

1. Standard Java imports for WebDriver.
2. We use the IT suffix for test in this book; this is the Maven convention for integration tests that run using the Failsafe plugin [1].
3. Tell web driver via this system property the location of the driver binary.
4. Create a new driver which connected to an instance of the Chrome browser.

5. Make sure that the browser quits when the test finishes.
6. Open a web page in the browser.
7. Locate an element on the current page, in this case the page's heading.
8. Verify that the heading is the value you expect.

You'll need to start up the test project as shown in the previous section before you run the test. Then, when you run the test, you should see the browser open a page similar to figure [Hello WebDriver](#).

Summary

- You can use WebDriver to save time and money by automating browser tasks.
- It is especially suited to automated browser testing.
- WebDriver is built around a standard JSON protocol, and that means all major browsers and languages support it.
- There are some great reasons to use WebDriver over manual testing. For example, you can save costs and improve quality at the same time.
- You need some tools to get started. We'll be using Maven and Java in this book.

In the next chapter we will start out on our journey by looking at the first part of any automation script—locating elements on pages.

1. <https://maven.apache.org/surefire/maven-failsafe-plugin/>

Chapter 2: Locating elements on a page

This chapter covers

- Locating elements using the By locators
- Writing your own locator factories
- Best practices for choosing and organizing locators

As a user, the first thing you do when opening a web page is try to find the part of the page you're interested in. If it is our social network, this might be the new messages icon; if it is our online banking, this might be our balance. You then use that information to consider your actions. In the same manner, WebDriver provides a wide variety of methods to locate elements on a page. This allows us to examine the page and enables us to check whether we see what we expect.

Without the ability to find those elements, it's not possible to do any interaction with the page. This makes the techniques necessary to do so incredibly important.

In this chapter we'll cover all the main methods for locating elements on pages. We will also look at how to compose those basic building blocks together into more complicated locators. Locating elements is only half the story. We will also cover ways to manage the different locating strategies you might want to use to make your code friendly and easy to work with.

Unlike human perception, WebDriver's methods are very strict. If you're looking for the new messages icon, you might be looking for something that looks like an envelope, but if that envelope is one day bigger or smaller, or a different color than how you remember, you will adapt. WebDriver doesn't have that intelligence. If the icon changes from an envelope to something else, most users would quickly pick up on the change, but not every WebDriver invocation would. We will look at patterns you can use to make sure that your code remains robust to changes in the page that might otherwise break your code and create a maintenance issue.

Not only will we introduce you to all the main element location methods, we'll cover what each one is more suited for, as well as organization practices. Finally, we will look at how to make the site you are automating easy to automate.

As the end of the chapter you will have an excellent toolbox of robust strategies for locating elements of all types.

Locating one or more elements based on ID, class, or name

WebDriver provides a number of ways to locate elements. We'll look at how this works, and the main ways you can achieve this.

What is a search context?

Locating elements on any page occurs within what WebDriver refers to as a *search context*. Let's quickly look at the `SearchContext` interface:

```
public interface SearchContext {  
    List<WebElement> findElements(By by);  
  
    WebElement findElement(By by);  
}
```

This interface provides two methods: one that finds all the elements that match an instance of what is known as a locator, and a second that finds the first element that matches, or throws a `NoSuchElementException` if it is not found. That means the `findElements` method is useful for finding elements when you're not sure if they will be on the page, or if you want to make sure they're not on the page.

Both the `WebElement` and `WebDriver` interfaces extend `SearchContext`, so all drivers and elements will have these methods.

What is a locator?

A locator describes what you want to find, and there is a zoo of them. In Java, you create a locator using the `By` class. Let's look at an example of finding an `h1` heading element on the current page:

```
WebElement heading1Element = driver.findElement(By.tagName("h1"));
```

And, for completeness, finding all the paragraph elements in the current page:

```
List<WebElement> paragraphElement = driver.findElements(By.tagName("p"));
```

As discussed, there are a number of core locators you can use to find elements on a page.

Locator	Usage
Class Name	Locates elements by the value of the "class" attribute.
CSS Selector	Locates elements via the driver's underlying W3 CSS Selector engine. If the browser does not implement the Selector API, a best effort is made to emulate the API. In this case, it strives for at least CSS2 support, but it offers no guarantees.
ID	Locates elements by the value of their <code>id</code> attribute.
Link Text	Locates elements by the exact text it displays
Name	Locates elements by the value of their <code>name</code> attribute.
Partial Link Text	Locates elements that contain the given link text
Tag Name	Locates elements by their tag name
XPath	Locates elements via XPath

Locating by link text

Link text is the preferred locator for links. Figure [Login form](#)

<http://localhost:8080/login.html> shows the link on the test app's login page for forgotten passwords:

The screenshot shows a login interface. At the top is a large dark blue header with the word "Login" in white. Below it is a horizontal row of three input fields: "Email" (text), "Password" (password), and a blue "Login" button. To the right of the password field is a blue link labeled "Forgotten Password".

Figure 1. Login form <http://localhost:8080/login.html>

This is represented by the following HTML:

```
<a href="#" id="change-password" class="btn">Forgotten Password</a>
```

And this can be located using the following:

```
driver.findElement(By.linkText("Forgotten Password"));
```

Any part of the string “Forgotten Password” can be used as the parameter for this method:

```
driver.findElement(By.partialLinkText("Forgotten Password"));
driver.findElement(By.partialLinkText("Forgotten "));
driver.findElement(By.partialLinkText("en Passwo"));
```

You should be cautious when using `findElement` with this locator. You may find other elements that contain the same partial text, so this should not be used to locate a single element on its own. Naturally, you can use it to locate a group of elements using the `findElements` method.

Locating by class attribute

This locates elements by the value of the `class` attribute. This can be used only for those elements having a `class` attribute, but it is not a good selector to use with the `findElement` method. Class is used for styling pages, and as a result many elements are likely to have the same class. As `findElement` always returns the first element it finds, if the element you want is not in the first place, you won't be able to use this to locate it. Even if it is the first element now, if a developer adds a new element with the same class earlier in the page, it will return the newly added element, instead of the element you're trying to locate. This makes for brittle code.

In the login page in figure [Login form http://localhost:8080/login.html](http://localhost:8080/login.html), the “Forgotten Password” link has one CSS class: `btn`; you can use class name `btn` to locate it:

```
driver.findElement(By.className("btn"))
```

Locating by ID

Locating an element by its `id` attribute can be expressed as the following:

```
driver.findElement(By.id("change-password"))
```

If the site is built using JavaScript, ID is normally applied to important elements. IDs are meant to be unique, so if an element has an ID, it's usually the most accurate way to identify the element. While IDs can accidentally appear multiple times on a page (for example, due to a programming error), this is rare. Since they are often added to facilitate JavaScript code, if this mistake occurs, the JavaScript will usually be faulty too, and therefore this will be spotted early in development.

If ID is available, make it your first choice.

Locating by input name

This is a locator which locates elements by the value of the `name` attribute. Normally it can only be used to locate form elements built using: `<input>`, `<button>`, `<select>`, and `<textarea>`. Remember, if the same name is used for multiple elements on the same page, only the first encountered element will be returned. So before using this locator, you need to check whether the name is unique on the page. If it is, then it can be used; otherwise, other locators (or combination thereof) will need to be used.

On the login page, there is an email input:

```
<input name="email" class="form-control" placeholder="Email"/>
```

So the code you would need would be the following:

```
driver.findElement(By.name("email"));
```

Locating by element tag name

This locator finds elements by their HTML tag name. Since there are often many repeating uses of most tags, it is not often possible to use this method to locate a single element. But it can be combined with other locators to effectively locate elements. One time you will find it useful is for locating the page's heading, as there is usually only one of these:

```
<h1>Welcome to WebDriver!</h1>
```

```
driver.findElement(By.tagName("h1"));
```

Locating using CSS selectors

Alongside XPath locators, the CSS selector locator is powerful. You can use the CSS selector approach to find by ID for example:

```
#change-password
```

CSS selectors can be used to locate most of the other selectors. You can infer that if you can't find the element using a CSS selector, you won't be able find the element using any of id, class name, or name attribute locator.

Locator	CSS selector equivalent
By class name "the-class"	.the-class
By ID "the-id"	#the-id
By tag name "h1"	h1
By name "the-name"	*[name='the-name']

The purpose of a CSS selector is to mark part of a page for formatting, not for automation. It is reasonable for a developer to change every CSS selector on a page, especially when a site gets a new coat of paint in the form of a re-skin. This means that CSS selectors, while powerful, can be a somewhat brittle.

CSS selectors can be slower than other locators, and it is worthwhile bearing that in mind if you are thinking of using them. This book is focussed on teaching WebDriver, not on teaching CSS, so you can find out more about CSS on the Mozilla's web site: <https://developer.mozilla.org/en-US/docs/Web/CSS/Tutorials>.

CSS is a good choice if you cannot locate an element by name or ID.

Locating using XPath

XPath locators are the most complex selector to use. It requires knowledge in XPath query language, so if you're not fluent in that query language, you will find it difficult to find elements using XPath queries.

Let's look at an example usage of an XPath for this HTML:

```
<a href="#" id="change-password">Change Password</a>
```

```
driver.findElement(By.xpath("//a[@id='change-password']"));
```

You can use the Web Developer Tools on Chrome to figure out the XPaths and CSS selectors. The XPath you get from the tool is `//*[@id="change-password"]`, which can be used directly, as in figure [Copying an XPath](#).

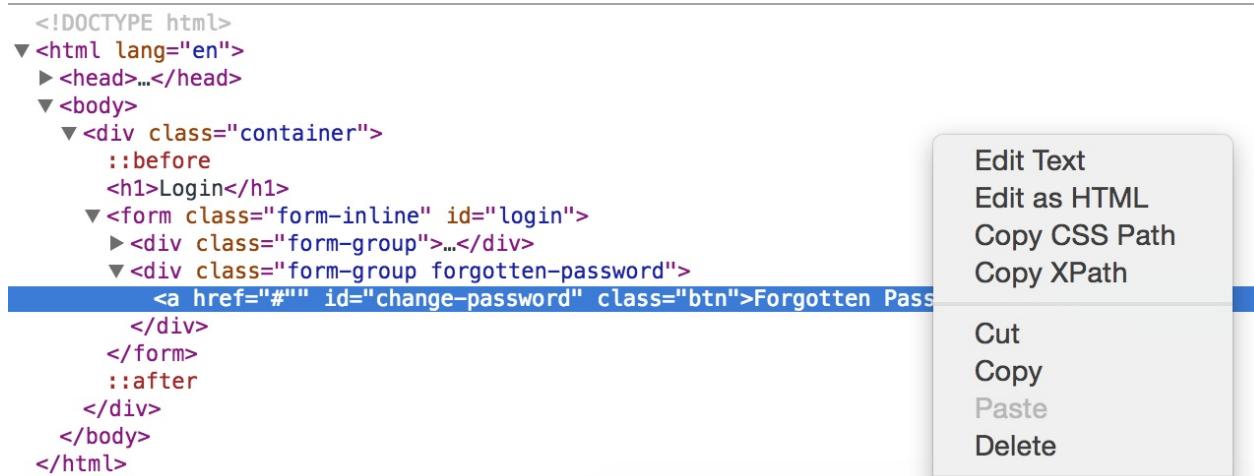


Figure 2. Copying an XPath

This finds any tag with an ID attribute of "change-password". As you know it is a link, so you can change it to `//a[@id='change-password']` and it will still work; this is also safer since it restricts the elements to tag `<a>`.

XPath and CSS selectors can fulfill some of the same goals.

Locator	XPath Selector Equivalent
By class name "the-class"	<code>//*[contains(concat(' ',normalize-space(@class),' '), ' the-class ')]</code>
By id "the-id"	<code>//*[@id='the-id']</code>
By tag name "h1"	<code>//h1</code>
By name "the-name"	<code>//*[@name='the-name']</code>

I hope you can see from this that you have the same power as CSS selectors—but the complexity is much higher.

Let's have a look at a couple of useful XPaths, ones that are hard to do using other locators.

normalize-space(.) versus text(.)

`text()` matches only against the text of the element, whereas `normalize-space()` returns the text of the element, and of all the elements it contains. This useful if you want to locate a paragraph of text which contains some styling:

```
<p>A paragraph with <b>this text in bold</b>.</p>
```

This will not be matched by the following XPath:

```
//*[contains(text(), 'A paragraph with this text in bold')]
```

Whereas this XPath will match:

```
//*[contains(normalize-space()), 'A paragraph with this text in bold'])
```

Here we have combined `normalize-space` with `contains`. You can use this to locate an element within a parent element.

```
//div[contains(., 'A visible paragraph')]/a
```

This is a powerful expression. It finds a link within a paragraph that contains the phrase "A visible paragraph." The main challenge with XPath expressions is their complexity. They take a while to learn, and as there is no type-safety, or auto-complete support in most IDEs, they can be costly to work with and maintain.

It should be noted that XPath is the slowest of all the locators. If you want to learn more about XPath, then you can do so at Mozilla's web site
<https://developer.mozilla.org/en-US/docs/Web/XPath>.

Many modern browsers provide a tool to find the XPath or CSS selector of an element. Be careful about copying these verbatim into your tests. They are often quite specific, and can easily change the next time the page loads.

XPath is good choice, but it's often better to try and solve locating problems using the CSS, name, or ID locators first, as these are easy for you and your team to understand.

That's a great variety of locators! They are all orientated around the knowing the string that you need to pass to create it. For XPath and CSS locators, these can be long and complex. We can reduce the complexity of our code by encapsulating commonly used locators into small factories that we can then reuse. For example, rather than having a locator for finding inputs using an XPath, it might be easier to create a locator specifically for the task of locating inputs.

Creating a locator factory for complex locators

You may find yourself creating locators that are similar to one another. For these common ones, you can reduce the verbosity of your code by having factories to create them.

XPaths are complicated to write and costly to maintain.

Create locator factories, similar to the methods on the `By` class, for complicated locators. My recommendation would be that because `By` methods are not specific to an element, create one generic factory for locating without a tag name, named `ElementBy`, and then create one each for any tag you want to locate. For example, for locating `input` elements by the text of the label they are annotated with:

[InputBy.java](#)

```
public final class InputBy {  
    private InputBy() {  
    }  
  
    public static By label(String labelText) {  
        return By.xpath("//label[contains(., " + labelText + ")]/input");  
    }  
}
```

You'll notice that the constructor is private to prevent instantiation of the class.

For locating any element by the text it contains:

[ElementBy.java](#)

```
public final class ElementBy {  
    private ElementBy() {  
    }  
  
    public static By partialText(String text) {  
        return By.xpath("//*[contains(normalize-space(.), " + text + ")]);  
    }  
}
```

The preceding factory classes allow you to encapsulate complex locators within a class and then reuse them at will. By following a naming scheme and style similar to the `By` factory, you can ensure that other developers will find them easy to pick up and ensure that adoption is easier.

Later on we will talk about strategies for grouping locators together.

Fine-grained targeting using complex CSS selectors

We discussed using CSS selectors earlier. Like XPath queries, CSS selectors are much more powerful than the basic locators. Let's look at a few ways to locate elements using CSS selectors.

You can refine a locator using an attribute:

```
<input name="email">
```

```
input[name='email']
```

Or multiple attributes:

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

```
input[type='text'][name='email']
```

Note that if the attribute is its default value, like `type` could be in this example, then its values will be undefined. You will not be able to locate it this way.

You can also refine a locator by the attributes prefix:

```
<input name="password">
```

```
input[name^='passw']
```

By the suffix:

```
<input name="password">
```

```
input[name$='sword']
```

By infix—containing a string:

```
<input name="password">
```

```
input[name*='sswor']
```

By the next element on the page:

```
<input name="password">
<input type="submit">
```

```
input[name='password'] + input[type='submit']
```

Or more loosely, by any sibling element:

```
input[name='email'] ~ input[type='submit']
```

By a direct descendant:

```
<div>
  <input name="email">
</div>
```

```
div > input[name='email']
```

Or by any descendant:

```
<form>
  <div>
    <input name="email">
  </div>
</form>
```

```
form input[name='email']
```

You will notice that most of the examples here contains no more than two elements.

This is deliberate. Think about this long selector:

```
div > #login > div > input.btn.btn-primary
```

If you were to remove one of those divs, it would make the selector invalid. Consider this one, suggested by Chrome:

```
#login > div:nth-child(1) > input.btn.btn-primary
```

If the divs were reordered, then this would no longer be valid. Optimal, easy to maintain selectors are concise and accurate.

Locating table cells using CSS selectors

HTML tables are a common way to layout data on a web page. As you may wish to locate cells by the heading of the cell's column, then you need to do a series of look ups. Lets look at a technique to make accessing them less complex.

You want to robustly locate elements within a table, but tables can change, and columns can be re-ordered.

Locate table cells using CSS selectors. You can access cells within a HTML table using CSS selectors. Consider the table of users in figure [Users Table](#) <http://localhost:8080/users-table.html>.

Users

#	Email	Name
1	john@doe.com	John Doe
2	jane@smith.com	Jane Smith

Figure 3. Users Table <http://localhost:8080/users-table.html>

This is created by the following HTML:

[users-table.html](#) - <http://localhost:8080/users-table.html>

```

<table class="table table-striped" id="users-table">
  <caption>Users</caption>
  <thead>
    <tr>
      <th>#</th>
      <th>Email</th>
      <th>Name</th>
    </tr>
  </thead>
  <tbody> (1)
    <tr> (2)
      <td>1</td>
      <td>john@doe.com</td>
      <td>John Doe</td> (3)
    </tr>
    <tr>
      <td>2</td>
      <td>jane@smith.com</td>
      <td>Jane Smith</td>
    </tr>
  </tbody>
</table>

```

1. Within the table's `tbody`
2. 1st row
3. 3rd column

Here is a selector that will find "John Doe" in the table from figure [Users Table](#) <http://localhost:8080/users-table.html>:

```
table#users-table tbody tr:nth-child(1) td:nth-child(3)
```

CSS3 and HTML unit driver

If you are using HTML Unit Driver, you may find that examples that use CSS3 selector syntax (like `:nth-child`) do not work.

This selector finds the third cell of the first row of the table with the ID `users-table` body. This is typical of table access: find the cell at x/y coordinates. Note that the indexing is 1 to N rather than 0 to N-1. This selector has an equivalent XPath query:

```
//table[@id='users-table']/tbody/tr[1]/td[3]
```

Our first task is to find the column index of the column you want. Since you want to get a cell by the column title, you can do the following:

TableIT.java

```
int columnNumber = 1;
while (!driver
    .findElement(By.cssSelector(String.format("table#users-table th:nth-child(%d)",
, columnNumber)))
    .getText().equals("Name")) {
    columnNumber++;
}
```

This loop will break when it finds the header; otherwise it will exit with a `NoSuchElementException` if it can't find it. You can find the row based on the returned number.

```
By.cssSelector(
    String.format("table#users-table tbody tr:nth-child(1) td:nth-child(%d)",
        columnNumber))
```

This is a good first example of working with a section of a page's DOM, rather than with a single element. Certain page structures, such as forms and tables, are reasonably complex, and the relationships between the elements within them is important to locating the specific element you are interested in.

This is also a great opportunity to create a *locator factory*:

TdBy.java

```
public final class TdBy {
    private TdBy() {
    }

    public static By cellLocation(int rowNumber, int columnNumber) {
        return By.cssSelector(String.format("tbody tr:nth-child(%d) td:nth-child(%d)",
            rowNumber, columnNumber));
    }
}
```

You'll notice that you can use XPath and CSS based locators interchangeably. But when should you use XPath, and when should you use CSS selector? There are no hard and fast rules, but as XPaths are oriented around the structure of the document you are

looking at, and CSS around the styling, you can ask yourself the question: "Am I locating based on structure, or based on style?"

Next, lets look at an example of chaining search context together.

Narrowing down by locating within chained search contexts

You can combine search contexts together. For example, if you can locate a form within a page, you can then easily find the submit button for that form by using the `findElement` method of the form. But if you need to search the whole page, you might have many buttons. In this technique we will look at how we can chain searches together to find an element easily and accurately.

You have an element that is hard to find with a single locator.

`WebElement` implements `SearchContext`. This means you can chain calls together to narrow down the context until you find your element. Let's have a look at the raw HTML of the login page:

[login.html - http://localhost:8080/login.html](#)

```
<div class="container">
    <h1>Login</h1>

    <form class="form-inline" id="login">
        <div class="form-group">
            <input name="email" class="form-control" placeholder="Email"/>
            <input type="password" name="password" class="form-control"
placeholder="Password"/>
            <input type="submit" value="Login" class="btn btn-primary"/>
        </div>
        <div class="form-group forgotten-password">
            <a href="#" id="change-password" class="btn">
                Forgotten Password</a>
        </div>
    </form>
</div>
```

The `btn` class is used twice here. To find the "Forgotten Password" link, you can chain two `findElement` calls together to find the precise one you want:

[ChainedLocatorsIT.java](#)

```
driver
    .findElement(By.className("forgotten-password")) (1)
    .findElement(By.tagName("a")); (2)
```

1. Find a form group for the forgotten password.
2. Find a link in that group.

You have a number of ways to search for common elements. If the element is harder to find, then you can use a chain of `findElement` invocations to narrow down the search context until you have the element you want. This is really effective when working with forms. You can find the form with a carefully created locator, then be a bit looser finding the elements you want within the form:

[ChainedLocatorsIT.java](#)

```
WebElement loginForm = driver.findElement(By.id("login")); (1)
WebElement emailInput = loginForm.findElement(By.name("email")); (2)
WebElement passwordInput = loginForm.findElement(By.name("password"));
WebElement submit = loginForm.findElement(By.className("btn-primary"));
```

1. Locate the form itself.
2. Narrow down each element.

This can resolve a number of issues with brittle location code.

Locator composition

You have seen that we repeatedly perform the same operation: find an element within a page, then narrow down the scope to find another element. You have seen this with forms and tables. What you should try to do is to keep locators simple: find a cell within a table, or an input within a form. You can compose them together to provide more complex locating.

WebDriver does not provide this out of the box, but the support library has a number of useful locators.

WebDriver Support Library

WebDriver provides a small support library. This contains a number of useful classes that are not part of the core library. These classes include support for (amongst other things) select lists, and Page Objects. These are well worthwhile including in your

project.

Locating by ID or name

This locator finds by the value of either ID or name attribute. This is useful for situations where you expect a page to change, and you want to be robust to that change. This is especially useful if the automation code doesn't live with the production code, and you can't guarantee they will get updated in step with one another. It is an example of *backward/forward compatibility*. It means you can update your test before the change is made, and it will still pass both before and after the change has occurred.

Consider this password input:

```
<input type="password" name="password" class="form-control"/>
```

It might be that the developers need to change the name to `j_password`. You can negotiate with them to add an ID to it:

```
<input type="password" name="password" id="password" class="form-control"/>
```

Then you can use the following locator:

```
driver.findElement(new ByIdOrName("password"));
```

This will work with before and after the change has been made. One thing to note is that if you use `ByIdOrName` with `findElements` (plural) you will get the super-set of elements that match: all elements that have the ID and name.

Chained locators

This locator performs a sequence of matches, drilling down into the DOM to find the element. It is probably best explained using an example of finding the email input within the registration form, but without using any major attribute of the email input:

```
<form role="form" id="registration-form"> (1)
  <div class="form-group">
    <label> (2)
      Email
      <input type="email" name="email" class="form-control"
        placeholder="E.g. john.doe@swb.com"/> (3)
    </label>
  </div>
<!-- ... -->
</form>
```

1. Find form by ID "registration-form"
2. Find a label with the text "Email"
3. find an `input`

LocatorCompositionIT.java

```
driver.findElement(
  new ByChained(
    By.id("registration-form"),
    By.xpath("//label[contains(.,'Email')]"),
    By.tagName("input")
  )
);
```

The example in listing [LocatorCompositionIT.java](#) will first find the registration form by it's ID, then find the label within that form, and finally the input within that locator. Each locator is executed with the `WebElement` returned by the previous element. The preceding code is equivalent to this:

```
driver.findElement(By.id("registration-form"))
  .findElement(By.xpath("//label[contains(.,'Email')]"))
  .findElement(By.tagName("input"));
```

Chained locators are useful when you want to mix and match different locators.

Creating locator composers to provide backward/forward compatibility

Imagine a new version of your web site is going to be released. But the HTML changed so that your locators no longer work – frustrating! You could update the locator to the new version, but that might not be complete yet, or you could leave your tests "red" and make sure everyone knows that they need to be updated at some point in the future. Or, you could make sure your test pass with both the old and the new code. This technique will show you how.

You want to locate elements in a reliable and flexible fashion.

Create a locator factory that composes locators using matching operations such as `all`, `any` and `none`. For example, you can create an "all" locator as follows:

AllBy.java

```
public class AllBy extends By {  
  
    private final By[] bys;  
  
    private AllBy(By... bys) {  
        this.bys = bys;  
    }  
  
    public static AllBy all(By... bys) { (1)  
        return new AllBy(bys);  
    }  
  
    @Override  
    public List<WebElement> findElements(SearchContext context) {  
        List<WebElement> elements = null;  
        for (By by : bys) {  
            List<WebElement> newElements = context.findElements(by);  
            if (elements == null) {  
                elements = newElements; (2)  
            } else {  
                elements.retainAll(newElements); (3)  
            }  
        }  
        return elements;  
    }  
}
```

1. Static factory method.
2. If you have the first set of elements, initialize the list.
3. Otherwise, keep only the new elements.

You can use this as follows:

LocatorCompositionIT.java

```
driver.findElement(By.allOf(By.tagName("input"), By.name("password")))
```

You can emulate the `IdOrName` locator with a `AnyBy`:

AnyBy.java

```
public class AnyBy extends By {  
    private final By[] bys;  
  
    private AnyBy(By... bys) {  
        this.bys = bys;  
    }  
  
    public static By any(By... bys) {  
        return new AnyBy(bys);  
    }  
  
    @Override  
    public List<WebElement> findElements(SearchContext context) {  
        List<WebElement> elements = new ArrayList<>();  
        for (By by : bys) {  
            elements.addAll(context.findElements(by)); (1)  
        }  
        return elements;  
    }  
}
```

1. Add all the elements you find.

This can be used as follows:

LocatorCompositionIT.java

```
driver.findElement(AnyBy.any(By.id("email"), By.name("email")))
```

If you want to exclude matches, the code is straightforward:

NotBy.java

```
public class NotBy extends By {  
  
    private final By by;  
  
    private NotBy(By by) {  
        this.by = by;  
    }  
  
    public static By not(By by) {  
        return new NotBy(by);  
    }  
  
    @Override  
    public List<WebElement> findElements(SearchContext context) {  
        List<WebElement> elements =  
            context.findElements(By.cssSelector("*")); (1)  
        elements.removeAll(context.findElements(by));  
        return elements;  
    }  
}
```

1. Get every element on the page.

For example, to find all contact check boxes on the registration page, but excluding the email box:

[LocatorCompositionIT.java](#)

```
driver.findElements(  
    AllBy.all(  
        By.name("contact"),  
        NotBy.not(By.cssSelector("*[value='email']"))  
    )  
);
```

The hawk-eyed among you will have noticed the similarity to Hamcrest matchers. This is deliberate; it will make them easy to learn. The question you might want to ask is "*Why doesn't WebDriver come with these bundled?*" Performance? Perhaps. The `NotBy` will need to return every element on the page. This might be quite slow, especially when working with a Selenium grid, or on pages with large numbers of elements.

This is just one way to combine locating elements in a complex way.

Next, we will look at a way to make pages easier to work with.

Making pages amenable to element locating

One of the interesting challenges to page automation is that we are automating an application which we do not write ourselves. I think my biggest recommendation to automation is: if you can, get write-access to the source code of the application you're automating. If you find that there is an element that you can't locate without a complex selector, you can modify it to make it easier to work with.

Item	Usage
ID attribute	JavaScript and CSS.
Name attribute	Form parameters.
Class attribute	Styling page.
CSS	Styling the page.

Any one of these can be legitimately changed by their author. Nowhere in the list does it say "used for automation".

Using a common CSS class prefix to create your own automation namespace

The HTML of the web site is designed for making the web site look good and customers happy. But what about our test automation? We can please everyone! This technique will show you how to mark up the HTML in such a way that changes to the HTML needed for styling or anything else won't affect your tests.

Pages change causing automation code to break.

Get write access to the page source code. Add automation data to it. Revisiting the HTML for the shopping cart:

[shopping-cart.html](#)

```
<input type="text"
      name="cartDS.shoppingcart_ROW0_m_orderItemVector_ROW0_m_quantity"
      class="form-control input-sm" value="1" size="2"/>
```

The one aspect we can change without affecting the behavior of the page is the class name. We can append a new, special class to it, in this case `wd-cart-item-0`:

```
<input type="text"  
      name="cartDS.shoppingcart_ROW0_m_orderItemVector_ROW0_m_quantity"  
      class="form-control input-sm wd-cart-item-0" value="1" size="2"/>
```

To break this down into its parts:

`wd`

An *automation prefix* ("wd" for WebDriver, but you might want `auto` for automating).

`cart-item-0`

An *automation ID* for us to access.

We can use this as follows:

```
driver.findElement(By.className(String.format("wd-cart-item-%d", 0)))
```

This technique relies on having access to the source code, and a clear and well-known agreement that classes prefixed with your automation prefix are only for automation, and can't be used for programming or other reasons. It would be easy for a new developer to accidentally undermine this technique if they don't know about it, so it's very important that everybody be aware of it.

Rather than have the common prefix littered across your code, this technique can be combined with locator factories to produce more concise code, for example:

ElementBy.java

```
public final class ElementBy {  
    ...  
  
    public static By automationId(String id) {  
        return By.className("wd-" + id);  
    }  
}
```

Used as follows:

```
driver.findElement(ElementBy.automationId(String.format("cart-item-%d", 0)))
```

When to use each locator?

We've covered both the built in locators, as well as some customized ones. But when to use each one? Here's a handy cheat-sheet to help you!

Locator	When to use it
ID	When your element has a unique ID
Name	When you're testing a form, and the input's name is unique
Class name	When the element has a unique class name
Tag name	When you want to locate a page heading
Link text	When you need to locate a link, and only care about one Language
CSS	When you need a complex locator, or you have an automation prefix
XPath	When you need a complex locator, and CSS will not work
Chained	When you need a complex locator, but want to mix different types of locator

Summary

- `SearchContext` is the main interface for locating elements, the methods `findElement` and `findElements` the primary methods.
- You have several different methods of locating elements. These include by ID, name, class, CSS, and XPath.

- Both XPath and CSS locators are powerful. This power can produce complexity, and therefore using patterns, such as composition and the strategy pattern, can greatly help to simplify their usage.
- Creating *locator factories* can make complex locators safe to create.
- You can encapsulate complex page structures, such as tables, into helper classes. The Decorator Pattern is useful when you are doing this.
- Hard to find elements can be located using the **navigational pattern**.
- If you have access to the site you are automating, an *automation prefix* can reduce the complexity of automating pages.

Now that you understand how to locate elements within a page, it is time to move forward with covering the various methods of interacting with those elements. The focus of the next chapter will be working with the various methods that `WebDriver` and `WebElement` provide to interact with the page.

Chapter 3: Interacting with elements on a page

This chapter covers

- Entering text into input fields, text areas
- Working with the mouse to click on buttons and links
- Working with touch

When automating interactions with a page, you are taking on the role of the user. That means that you will want to be able to do everything the user can do, including entering text from the keyboard and selecting and interacting with individual elements on the page. WebDriver provides a number of ways to do this, but for most cases you'll only need to use a subset of them, so in this chapter we will first cover the most common, and therefore most useful, methods.

In this chapter you'll learn how to:

- Enter text using both the `WebElement` and `Keyboard` classes
- Control the mouse using the `WebElement` and `Mouse` classes.
- Perform specific techniques for working with forms.

Be the user

When crafting a test to simulate the behavior of a user, it is worthwhile putting yourself in the place of the user, and then breaking down the steps you would take. I like to consider what goal that the user wants to achieve, and what operations I'd undertake to achieve that goal. For example:

My goal would be to login to a site so I can send an email.

The operations could be:

1. Click on the input labeled “Username.”
2. Enter the username.

3. Click, or tab, to the input labeled “Password.”
4. Enter password.
5. Either, click the Login button, or press Return.

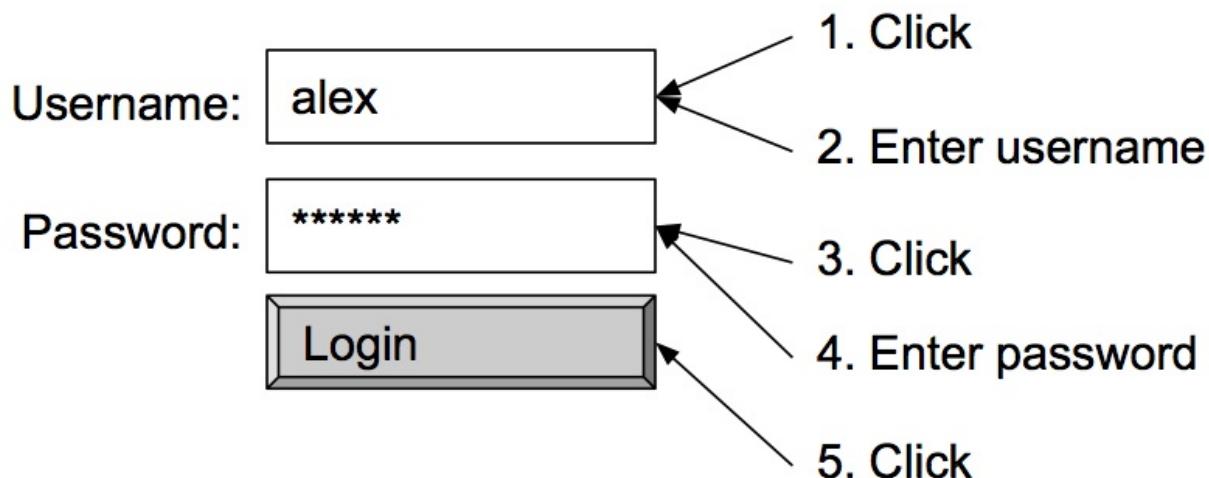


Figure 1. Login Form <http://localhost:8080/login.html>

It is worthwhile noting that the user may be doing a few things that are not immediately clear. For example, when entering a password, they might check that the password’s text is obscured, or they may check that the page is secure. We will cover the verifications that a user might do in chapter 7.

One of the main things a user will be doing, when they complete forms, is entering text into those elements of the form.

Entering text

Entering text is key to a number of common tasks when testing web pages. The most common would be completing a form. Think of the tasks the user might want to achieve:

- Log in or register on a site.
- Send a social network update.
- Complete the purchase of an item with a credit card.
- Compose an email.
- Update a blog post.

The text you might need to enter might be plain text, but you may need to enter rich-text, text that maybe contain bold or italics for example.

WebDriver provides two ways to enter text. The first method is to directly send keys to an element on the page. This is convenient, as it is available as a method of the `WebElement` class. The second method, for drivers that implement the `HasInputDevices` interface, is to control the keyboard object. This is more powerful, allowing you to do more things, but is also more complex.

Let's look at the two primary methods that WebDriver provides for keyboard interaction.

WebElement.java

```
public interface WebElement extends SearchContext {  
  
    void sendKeys(CharSequence... keysToSend);  
  
    void clear();  
  
    ...  
}
```

And we can use this test to understand how `sendKeys` and `clear` work. Without clearing the text first, `sendKeys` will just append the text to the end of original text,

LoginIT.java

```
@Test  
public void login() throws Exception {  
    driver.get("/login.html");  
  
    WebElement email = driver.findElement(By.name("email")); (1)  
    email.clear(); (2)  
    email.sendKeys("john@doe.com"); (3)  
  
    WebElement password = driver.findElement(By.name("password"));  
    password.clear();  
    password.sendKeys("secret");  
  
    driver.findElement(By.cssSelector("input[type='submit']"))  
        .click();  
}
```

1. We locate the element first

2. We need to clear the text inside the input, otherwise it will append the text to original text
3. Set the string "john@doe.com" to the "email" field

Sending keys directly using the `WebElement` method has the advantage of being concise, and therefore is a good default choice for entering text in most cases. But, it lacks the precision of the `Keyboard` class. For example, you cannot control when a key is pressed and released.

You use an element locating technique to find the element you want, and WebDriver will send the text you want directly to it, as if you had selected the element with a mouse click and started typing. Unfortunately, you can't do everything a user can do with the keyboard using send keys.

You can also get a handle to the keyboard using the `getKeyboard` method of drivers that implement the `HasInputDevices` interface:

Keyboard.java

```
public interface Keyboard {  
  
    void sendKeys(CharSequence... keysToSend);  
  
    void pressKey(CharSequence keyToPress);  
  
    void releaseKey(CharSequence keyToRelease);  
}
```

The `Keyboard` interface allows you to simulate the exact actions of the user, down to the exact keys pressed. Using the `Keyboard` interface has the advantages of control and precision. You can do everything the user can do. Unlike the `sendKeys` method, you don't send key presses to a specific element. Instead they will be typed into the active element. You can easily type non-printable characters, including modifier keys (Alt, Ctrl, Command, and the Japanese Zeukauk-Hankaku), function keys (F1 to F12), Backspace, and arrow keys (Left, Right, Up, and Down). If you have a complicated application, such as a web-based game, then you may well need to use these. If you want to find out more information on printable and non-printable characters:

<http://en.wikipedia.org/wiki/ASCII>. The `Keyboard` interfaces doesn't provide support for element location; the text you type is sent to the currently active element, this means you will need to focus the element.

To focus an element either send an empty string to it:

```
element.sendKeys("");
```

Use the actions class to move the mouse to it:

```
new Actions(driver).moveToElement(element).perform();
```

And for an input element, click on it:

```
new Actions(driver).moveToElement(element).click().perform();
```

One interesting thing about the keyboard is that typing a single letter can be broken down into two smaller actions: pressing a key down, and then releasing it. This is useful if find yourself needing to test a page where the user will need to hold down the arrow keys.

On desktop browsers, it is possible to navigate the Internet using the keyboard alone. But it is usually slower than using a mixture of mouse and keyboard. This means that a user might use the mouse to select something that can have text entered into it, and then switch to the keyboard. On mobile devices, the keyboard may only appear when you tap on a text input.

Using sendKeys and getKeyboard to enter text into a form

Probably the most common reason to need to enter text is to complete a form. As mentioned before, users will often do this by using the Tab key to move between elements, colloquially known as *tabbing*. This following technique will show you how to enter text into a form on a page.

You want to test the behavior of a form on a page to make sure that it can be filled out primarily using the keyboard.

To demonstrate various methods, we will:

1. Use `findElement` to find the first text input.
2. Use `sendKeys` to enter text into the element.
3. Use `getKeyboard` to get the keyboard.

4. Enter Tab via the keyboard to tab to the next element.
5. Ask WebDriver for the active element.
6. Enter text using the keyboard.
7. Finally submit the form by pressing, then releasing, Return.

The example application contains a form for signing up to the mailing list, as you can see in figure [A mailing list sign-up form \(`http://localhost:8080/mailing-list.html`\)](http://localhost:8080/mailing-list.html).

Sign-up To Our Mailing List

Email

E.g. john.doe@swip.com

I accept the terms and conditions

Sign-up

Figure 2. A mailing list sign-up form (<http://localhost:8080/mailing-list.html>)

You can see that you need to enter your email address, check a box to accept the terms and conditions (which you will do using the spacebar), and submit the form (by pressing Return).

KeyboardInputIT.java

```
driver.get("http://localhost:8080/form.html");
driver
    .findElement(By.name("email")) (1)
    .sendKeys("john.doe@swb.com"); (2)

driver.getKeyboard().sendKeys(Keys.TAB); (3)
driver
    .switchTo().activeElement() (4)
    .sendKeys(" "); (5)

driver.getKeyboard().pressKey(Keys.ENTER); (6)
driver.getKeyboard().releaseKey(Keys.ENTER);
```

1. Locate the email input.
2. Enter the text into the input.

3. Now tab to the check box.
4. Change to the currently active element, in this case the check box.
5. Press space to check the box.
6. Submit the form.

This code demonstrates three ways you can interact with elements using the keyboard:

1. You can act on an element you have already found using the `sendKeys` method.
2. You can use the `Keyboard` object using the `getKeyboard` method, pressing individual keys.
3. You can use `WebDriver` to access the active element and type into it by using the `sendKeys` method. This is useful if you want to verify that the element tabbed to is the expected one.

The hawk-eyed among you will have noticed that we have used *method chaining* [1] to reduce the amount of code. I find that, if there is one suitably indented method call per line, it makes code easier to read.

To be able to effectively test web pages, you need to be able to complete the same actions a user would complete. `WebDriver` provides more than one method to do this, each of which are suitable for different tasks, but all make it easy to enter text into a page.

One final note is that, not all implementations of driver support the `Keyboard` class. Notable the classes `HtmlUnitDriver` and `SafariDriver` (two alternatives to `FirefoxDriver`) do not currently support it.

In summary:

Send Keys:

- Single command.
- Supported by all drivers.

Keyboard:

- Make element active then type.
- Support for all keys.
- Support for press then release.

- Not supported by all drivers.

Entering text into a WYSIWYG editor

Many forms now allow you to enter text into inputs that allow the user to change the style of the text being entered. For example, you may be able to make text bold or italic. A good example of this is word processor that is part of Google Docs, and in fact most web based email client allow you to do it too. This is more complex than normal forms, as not only do you need to type the text, you also need to tell the WebDriver how to switch to the appropriate styling. WYSIWYG editors (like Tiny MCE, which we will be using in our example) are not normal form inputs. They are JavaScript applications that modify the page typically doing the following:

1. Hide the original input.
2. In the place of the original input, insert a div containing controls (such as bold or italic).
3. In that div, insert an inline frame (iframe) into the page.

In fact, unlike other page inputs, you don't modify the input within the form at all. Instead, the iframe contains a page that is marked as *content editable*. A page that is content editable can be modified by the user.

You want to enter styled text into a WYSIWYG editor similar to figure [WYSWYG editor](http://localhost:8080/wyswyg-editor.html) (<http://localhost:8080/wyswyg-editor.html>).

WYSWYG Editor

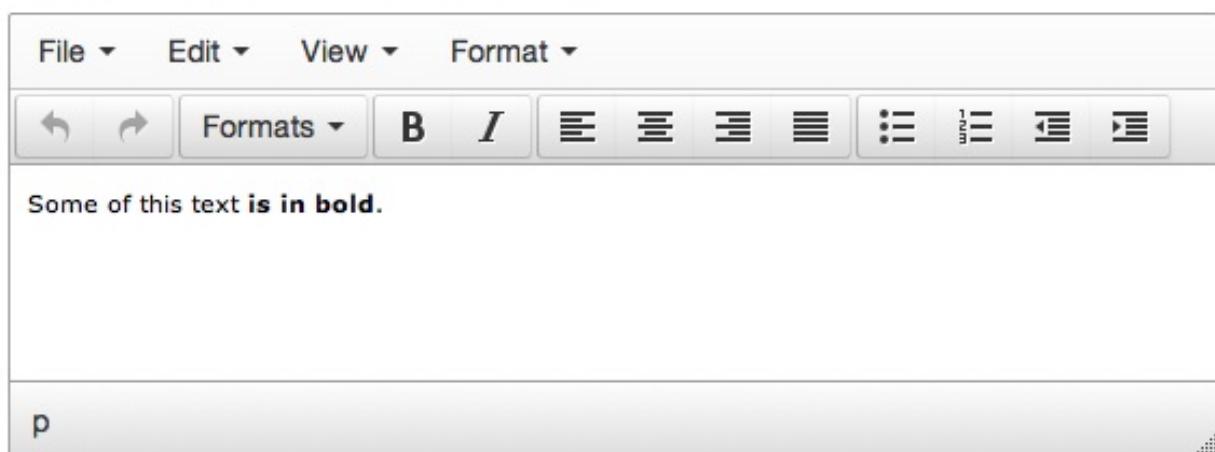


Figure 3. WYSWYG editor (<http://localhost:8080/wyswyg-editor.html>)

To enter text into this WYSWYG form you will:

1. Locate the frame that contains the editor.
2. Switch to that frame.
3. Locate the HTML body of the frame.
4. Choose italic.
5. Enter text.
6. Switch back to the default frame.

WyswygInputIT.java

```
driver.get("http://localhost:8080/wyswyg-editor.html");

WebElement editorFrame = driver.findElement(By.id("editor_ifr")); (1)

driver.switchTo().frame(editorFrame);

WebElement body = driver.findElement(By.tagName("body")); (2)

body.clear(); (3)
body.sendKeys("A paragraph of text, some of which is"); (4)

driver.switchTo().defaultContent(); (5)
driver.findElement(By.cssSelector(".mce-i-italic")).click(); (6)

driver.switchTo().frame(editorFrame); (7)
body.sendKeys(" italics."); (8)

driver.switchTo().defaultContent(); (9)
driver.findElement(By.cssSelector(".mce-i-italic")).click(); (10)
```

1. Find the correct iframe.
2. Get the body element.
3. Clear the existing text.
4. Send our unformatted text.
5. Switch back to the original frame.
6. Click the italic button.
7. Switch back to the editor.
8. Type italic text.

9. Switch back to root window.

10. Deselect italics.

Directly using WebDriver to interact with a WYSIWYG editor to complete even this basic task is complex. You have to deal with switching between frames, clicking a button to enable and disable italics, and coordinate both of these actions with entering text. All the items have class name or IDs that might change if you upgrade your version of the editor. There is also a fair amount of repetition.

Simulating user behavior using the mouse

While users can navigate and interact with a page using the keyboard, much of the time users will use the mouse instead. It's the most common way to interact with web pages. You will find yourself doing most of your automation by simulating the use of the mouse.

A number of operations can be done with the mouse. Not only can you click, you can double-click, right-click to open the context menu, move the mouse, and drag-and-drop. A web application can listen to all these using JavaScript, and so WebDriver allows you to simulate them all.

One great thing about the mouse is even though mobile devices don't have a mouse, mobile browsers will treat taps as if they were mouse clicks. This means you can use a lot of the testing methods you are familiar with on desktop for mobile testing.

A story from the trenches

While savvy users will happily navigate a page using the mouse, I've seen behaviors you might not expect from less experienced users. For example, I was showing a user how to submit a form on a new application. I said, "To finish your purchase, click the Buy button," they replied, "Left or right click?" Later on the same day, another user attempted to complete the same purchase by double-clicking the Buy button, when you would expect most users to single-click.

Let's have a look at the WebDriver methods at your disposal:

WebElement.java

```
public interface WebElement extends SearchContext {  
  
    void click();  
  
    ...  
}
```

That is only a single method. No problem. Web drivers that implement the `HasInputDevices` interface provide a `getMouse` method that returns a `Mouse` object. That object provides useful methods:

Mouse.java

```
public interface Mouse {  
  
    void click(Coordinates where);  
  
    void doubleClick(Coordinates where);  
  
    void mouseDown(Coordinates where);  
  
    void mouseUp(Coordinates where);  
  
    void mouseMove(Coordinates where);  
  
    void mouseMove(Coordinates where, long xOffset, long yOffset);  
  
    void contextClick(Coordinates where);  
}
```

You can see that there are quite a variety of methods here. Note that they all require a page coordinate to use. If you want to use this directly, you need to phrase your tests as:

1. Find coordinate of element.
2. Execute an action to that coordinate.

This is all rather laborious, when you just want simply click on an item. We will discuss another approach shortly.

Clicking and double-clicking with the mouse

The second main way to interact with a page is by clicking on elements using the mouse. In this technique you'll see the two main ways you can achieve this using WebDriver, and meet the `Actions` class.

You want to simulate the behavior of a user navigating a form using a mixture of keyboard and mouse. Specifically, you want to see what happens if the user double-clicks the Submit button: does the form get submitted twice?

WebDriver provides methods to click elements in a similar manner to sending key presses to an element. You will use the same mailing list form from figure [A mailing list sign-up form \(`http://localhost:8080/mailing-list.html`\)](#), where you will single-click on the check box, using this method. Here is an example of clicking on an element:

Clicking terms

```
driver
    .findElement(By.name("terms"))
    .click();
```

To double-click the submit button, you could use the `Mouse` class (though we'll show you a better way shortly):

Double-clicking

```
WebElement submitButton = driver.findElement(By.tagName("button"));

driver.getMouse().doubleClick(
    ((Locatable) submitButton).getCoordinates()
);
```

There are a couple of problems with this approach. The first problem is the class cast to make the element Locatable. The second problem is that `Locatable` is an internal WebDriver class, so it may change, or even be removed, in future versions of the WebDriver API. If we want low-maintenance tests, then we need to avoid it.

Good news! There is a special way for doing this kind of interaction: the `Actions` class. The `Actions` class allows you to create sequence of actions, which are then performed one after another. One of these possible actions is double-clicking, so you can create a sequence containing with that action.

Actions

```

new Actions(driver) (1)
    .doubleClick(submitButton) (2)
    .perform(); (3)

```

1. Create actions object from the driver.
2. Add a double-click to the sequence.
3. Perform the sequence.

Finally, here is the complete test:

[MouseInputIT.java](#)

```

driver.get("http://localhost:8080/mailing-list.html");
driver
    .findElement(By.name("email"))
    .sendKeys("john.doe@swb.com");

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

WebElement submitButton = driver.findElement(By.tagName("button"));

new Actions(driver)
    .doubleClick(submitButton)
    .perform();

```

Should you refactor this code by extracting a double-clicking method so you can reuse it elsewhere? Maybe. You should extract commonly used code, but as double-clicking is a less common action, perhaps wait until you have done it a few times.

With WebDriver, there are two ways to simulate mouse clicks: either the `click` method, or using `Actions`. You will notice that both `sendKeys` and `click` are available for elements, though most elements can only accept text, or mouse clicks, not both.

Context menus

On many applications you can use the right-mouse button to open a context menu. This menu will display a series of options related to the specific element you are clicking on. For example, in a word processor, right-clicking might open a series of

formatting options. In a file management application, right-clicking on the icon for a file might provide options such as “move” or “new directory.”

Figure [Context menu http://localhost:8080/context-menu.html](http://localhost:8080/context-menu.html) shows our context menu from the test application.

Context Menu



Figure 4. Context menu <http://localhost:8080/context-menu.html>

Opening a context menu

Context menus are rare in most modern web applications, perhaps because users expect the standard web browser menu to appear when the user right-clicks. However, you may still encounter them. This technique shows you how to open a context menu.

An application contains a context menu and you want to open it.

Locate the element and use `Actions` to perform a context click.

ContextMenuIT.java

```
driver.get("http://localhost:8080/context-menu.html");

new Actions(driver)
    .contextClick(driver.findElement(By.id("hascontextmenu")))
    .perform();
```

We can see that for complex operations, the `Actions` class comes in very useful. Context clicks are a bit of a rarity; you won't find them in many web pages, but web based applications such as email, word processors, or spreadsheets will have them.

Safari support

Currently, the Safari driver does not support context menus.

We'll look at the `Actions` class in much more detail in part 2 of the book.

Interacting with forms

There are a number of challenges when interacting with a form. For example, entering text into a `textarea` is quite different to an `input`. Choosing an option from a select list is not a single operation. The form in figure [Registration form](#) <http://localhost:8080/registration-form.html> shows most of the common elements that can be found in a form. They are, in order:

1. A normal input.
2. A password input.
3. A select drop-down.
4. A radio button group.
5. A multi select drop-down.
6. A check box.

Registration Form

Email

E.g. john.doe@swip.com

Password

How did you hear about us?

-

▲ ▼

Contact me by email. Contact me by phone.

Please choose what you are interested in

Books

Music

Movies

I accept the terms and conditions

Sign-up

Figure 5. Registration form <http://localhost:8080/registration-form.html>

To enter text into forms, you can use all the techniques you've learned earlier. Now, we'll look into the different types of form element, and what special problems you might need to deal with.

Password

Passwords are inputs where you can't see the text that is entered, and you can use the same techniques as entering text. One issue you're likely to encounter is that certain browsers, such as Safari, will ask you if you want to save the password.

If you have access to the machine that the browser is running on, if it is your local machine for example, you can edit the setting to disable this.

If you do not have access to the machine, then this won't work. We can only modify what we have access to--the page. We can ask WebDriver to execute some JavaScript that modifies the page. The HTML standard has an attribute `autocomplete` that is intended to prevent auto-completion of forms (for example for sensitive data such as credit card numbers).

```
<input type="password" autocomplete="off"/>
```

Unfortunately, several browsers have stopped honoring this attribute. The “Save Password” alert only appears if there is a password on the form. Instead, we can use JavaScript to convert all passwords to normal text inputs. We've not spoken about the `JavascriptExecutor` interface before. This interface allows you to execute JavaScript on the current page. It is useful if you want to do something that WebDriver does not support, such as modify the page it make it easier to test. To use it, you need to cast your WebDriver to it.

FormIT.java

```
((JavascriptExecutor) driver).executeScript("Array.prototype.slice.call(" +
    "document.getElementsByTagName('input')).forEach(function(e){" +
        " e.type=e.type=='password'?text:e.type;" +
    "});");
```

This will prevent the alert from appearing. But, don't forget—anyone who can access your computer will be able to see the password!

Part 2 of the book contains a deep-dive into the `JavascriptExecutor` interface.

Radio button

If you have a group of radio buttons, each button will have the same name. This means that select the using `By.name(...)` won't find the correct element. Here are two ways around this. The first option is to use an XPath locator to select the element based on the text it is labelled with. This XPath locator finds a label that has the text “email”, and then finds the input within.

```
driver
    .findElement(By.xpath("//label[contains(.,'email')]/input"))
    .click();
```

The second option is to locate the radio-button based on it's value:

```
driver
  .findElement(By.cssSelector("input[name='contact'][value='email']"))
  .click();
```

You'll probably find the second option most reliable.

Single and multiple-choice select

To work with a select, you wrap it in a `Select` object. The `Select` class is part of the WebDriver support library. This class provides a number of features for working with select boxes, and one useful one is `selectByVisibleText`. If we want to select "Friend" in the following HTML:

`registration-form.html`

```
<select class="form-control" name="hearAbout">
  <option>-</option>
  <option>Friend</option>
  <option>Advert</option>
</select>
```

We can use the `Select` class as follows:

`RegistrationFormIT.java`

```
new Select(driver.findElement(By.name("hearAbout")))
  .selectByVisibleText("Friend");
```

`Select` can be used for single and multi-choice selects:

`RegistrationFormIT.java`

```
Select interestsSelect = new Select(driver.findElement(By.name("interest")));
interestsSelect.selectByVisibleText("Movies");
interestsSelect.selectByVisibleText("Music");
```

A form is a great candidate for a *page object*, so later on we will look into extracting this.

Summary

- You can use `sendKeys` method of `WebElement` to type into inputs. This is useful in the most common cases.
- You can use `Keyboard` for more complex interactions, but it is not as well supported.
- WYSIWYG editors require special techniques.
- You can click on elements using the `click` method of `WebElement`, and this will cover most common cases.
- If you want to double-click or right-click, you must use `Actions`.
- There are some techniques that are helpful with form elements which contain either password inputs, or select lists. You've seen that you can wrap an element in a `Select` object.

Now that you've read chapters 1 through 3, you have enough knowledge to find and interact with pages and the elements on them. In chapter 4 we will tie element location, and interaction, into verification so you not only can you automated interactions with the page, but verify that the actions worked correctly.

1. https://en.wikipedia.org/wiki/Method_chaining

Chapter 4: Examining a page

This chapter covers

- Checking to see whether an element is present and whether it's visible
- Verifying the text of headings and other elements
- Examining the CSS styling of an element

In chapter 2 we looked at locating elements, and in chapter 3 at interacting with them. But like a user, if you can't see what's on the page, you're blind. WebDriver provides a number of ways it can be your "eyes" when writing automation code.

Almost all testing requires you to put your system into a certain known state, perform some action on it, and then verify that the resulting state is what you expect. In *Behavior Driven Development*, [1] this is known as *given/when/then*. In the case of web automation, you navigate to a page, type or click elements of the page, and check whether the resulting page looks as you expect. WebDriver provides a number of ways to simulate the action of a user "looking" at the page.

In this chapter, we'll explain how to determine whether an element is visible on a page, and we'll look at some quick methods for checking page text. We'll also discuss using WebDriver's helper class `Select`. Then we'll examine how to verify attributes of an element, such as color and size. Much of this revolves around using methods of the `WebElement` class.

By the end of this chapter, you'll have learned how to write code that uses all the key tools WebDriver provides to "look" at a page.

Checking whether an element is present

For this chapter, we have styled the web page shown in figure [Screenshot of a styled page - http://localhost:8080/styled-elements.html](#). This page has common elements, and it also has some hidden ones.

This page contains a variety of styled elements.

A visible paragraph.

A paragraph with **this text in bold**.

Some red text.

Figure 1. Screenshot of a styled page - <http://localhost:8080/styled-elements.html>

One of the tasks you may want to do is to determine whether an element is present on a page. You'll doubtless find yourself in a situation in which you want to check that something isn't yet visible. For example, a online shopping basket might be invisible until you click a button. Fortunately, WebDriver can give you access to elements a user can't see. Let's look at how to do this.

Using `findElements` to determine whether an element is present

Some times you want to not just check the color or text of something on the page. You might want to check something more fundamental – whether or not it exists. This following technique will show you how to do that using the `findElements` method.

You want to check whether an element is present on a page or not.

The `WebElement` interface provides two method to do this. When an element may not be present, you should locate it using `findElements` (plural) rather than `findElement` (singular). The method `findElement` will throw an exception if the element can't be found, but `findElements` will return an empty list:

[ElementIsPresentIT.java](#)

```
@Test
public void checkingAnElementIsPresent() throws Exception {
    driver.get("http://localhost:8080/styled-elements.html");

    assertThat(driver.findElements(By.id("invisible")), hasSize(1));
}
```

You could use a few different approaches to check the elements in the list. You could check whether it's an empty list, or you could assert that its size equals 1. The problem with both of these methods is that when the test fails, you don't get clear diagnostic information. Instead, you might want to use `assertThat` (as we do in listing [ElementIsPresentIT.java](#)), as this typically gives more information.

Verifying whether an element is visible (or not)

Sometimes you may have an element that is hidden from view and that becomes visible only after some action, such as opening a navigation menu, and you don't wish to continue unless the element is visible. Or you may want to make sure an element isn't visible after some operation has occurred. `WebElement` provides the `isDisplayed` method to help you achieve this.

Using `isDisplayed` to determine whether an element is visible

You know that an element will be on a page, and you want to make sure it's visible.

Use `findElement` to find an element, and then check whether it's visible using `isDisplayed()`. Consider the following HTML:

```
<p id="invisible" style="display:none;">Some hidden text.</p>
```

The paragraph of text is invisible.

ElementIsVisibleIT.java

```
@Test  
public void visibleElementIsDisplayed() throws Exception {  
    driver.get("http://localhost:8080/styled-elements.html");  
  
    assertTrue(driver.findElement(By.id("visible")).isDisplayed());  
}
```

This test is shorter than a check for the presence of an element. It can be used to check that an item isn't visible. Naturally, it would be time-consuming to check for every hidden element on the page! This technique is most useful in the case when you want to make sure an element is invisible until some action is performed, as per listing [invisible].

```
assertFalse(driver.findElement(By.id("invisible")).isDisplayed());
```

Verify the page title

Checking the page title is straightforward and fast. This can make it a great first choice for checking that the correct page has been loaded after a link or button has been clicked.

Using getTitle to check the title of a page

Many parts of a page are subject to change based on how the page's developer lays them out. But one part is almost always available, is quick to access and fundamental to the page – its title. This technique will show you how to access and verify that.

You want to make sure a page's title is correct, but you want to ignore any prefix the title might have.

You can use the `getTitle` method to get the current page's title:

PageTitleIT.java

```
@Test  
public void checkThePageTitle() throws Exception {  
    driver.get("http://localhost:8080/styled-elements.html");  
  
    assertThat(driver.getTitle(), containsString("Styled Elements"));  
}
```

It's extremely straightforward to verify a page's title. Our test site prepends the site's name to every page title, so using `containsString` means you won't have to rewrite the test if the site's name changes. In figure [Page assertion error](#), you can see the diagnostics produced by using `assertThat`.

```
java.lang.AssertionError:  
Expected: a string containing "Styled Elements"  
      but: was "Test Application - Enter Title Here"
```

Figure 2. Page assertion error

Verifying that text is on the page

You'll often want to make sure certain text has appeared in response to an operation. For example, clicking the Buy button results in a Purchase Confirmed message. If you know where the text will be on the page, you can use `getText` method to verify it:

```
@Test  
public void checkItemWasPurchased() throws Exception {  
    ... // purchase code here  
  
    assertThat(driver.findElement(By.id("confirmation")).getText(),  
        containsString("Purchase Confirmed."));  
}
```

Checking text when you don't know where it might be on the page is more challenging. Let's look at a couple of approaches that work well.

Using XPath and an element stream to verify that text is on the page

Sometimes you might find that you're not too worried about where text is on a page, but more worried that it is actually visible and correct. This technique will show you how to use XPath to achieve that.

You want to verify that text appears on a page, but you don't know where it might appear.

If you can't find the element easily, you have two options. We'll cover each of these in turn.

Option 1: The *easy to find method* checks that the text is somewhere in the page source.

VerifyingTextIT.java

```
@Test  
public void pageSourceMethod() throws Exception {  
    driver.get("http://localhost:8080/styled-elements.html");  
  
    assertThat(driver.getPageSource(),  
        containsString("This page contains a variety of styled elements."));  
}
```

Unfortunately, this also recognizes your text in the header, a script, or even a page comment. Thus tests may pass or fail when they shouldn't. Consider the following example.

VerifyingTextIT.java

```

    @Test
    public void whenPageSourceFails() throws Exception {
        driver.get("http://localhost:8080/styled-elements.html");

        assertThat(driver.getPageSource(),
            anyOf(
                containsString("<p id=\"invisible\" style=\"display:none;\">"), (1)
                containsString("<p style=\"display:none;\" id=\"invisible\">")
            )
        );
        assertThat(driver.getPageSource(),
            containsString("<!-- a comment about the page -->")); (2)
    }
}

```

1. The order of an element's attributes may not be the same in different drivers.
2. This even passes on a page comment.

Option 2: The *XPath text method* uses a complex XPath to find the element.

VerifyingTextIT.java

```

    @Test
    public void xpathTextMethod() throws Exception {
        driver.get("http://localhost:8080/styled-elements.html");

        assertNotNull(driver.findElement(By.xpath("//*[text()='A visible paragraph.'])));
    }
}

```

You can see from the previous examples that there are several ways to check for text on a page. The best approach depends on your circumstances. We recommend targeting the element directly by ID or CSS locator, as shown in chapter 2.

When direct location fails, the page source method is concise but can produce false positives when the text isn't on the page—for example, it forms part of comments, script, or style tags. The page source method is also slow on large pages, because the page's entire source must be transferred from the browser to the driver each time you use it. Furthermore, the page source can be different for the same page in different browsers.

The XPath text method is concise, but the example only checks the entire element's text.

Examining an element's styling

In addition to verifying text you can see, you'll want to check that it's styled the way you expect. You can do this using the `getCssValue` method, which returns the CSS value of the style applied to the element.

Verifying an element's style using `getCssValue`

All elements on a page have styling. This can be important, for example, you might want error messages is red, and informational messages in blue. This technique will show you how to check an element's style, and some pitfalls to watch out for.

You want to make sure an element is styled correctly.

Checking the style of an element isn't straightforward. You can specify the color of a web page element via inlined-styles:

```
<p style="color:red;">Some red text.</p>
```

Or using CSS:

```
<p class="red">Some red text.</p>
```

Watch out! Although both paragraphs are colored with "red", when you access their color, you'll get what's known as the *computed value*. The computed value is the color of the element you see on the page, and it's the one you have to check.

Computed colors are usually in the format `rgba(red, green, blue, alpha)`, where `red`, `green`, and `blue` are integers between 0 and 255. Red is therefore `rgba(255, 0, 0, 1)`. You may find that some colors are returned without alpha, as in `rgb(red, green, blue)`:

VerifyingStylesIT.java

```
@Test
public void elementHasRedText() throws Exception {
    driver.get("http://localhost:8080/styled-elements.html");

    WebElement element = driver.findElement(By.id("red"));

    assertEquals("rgba(255, 0, 0, 1)", element.getCssValue("color"));
}
```

The `WebElement` class provides the `getCssValue` method to examine an element's CSS values. These aren't the values applied inline or via a stylesheet, but the computed values. This means they may not be the same and the source code of the web page. Watch out *shorthand values*, which aren't meant to be returned. A shorthand value is used to aggregate several independent values. For example, consider this piece of HTML:

```
<div style="border: solid #000 1px">  
    ...  
</div>
```

This is short-hand for

```
<div style="border-style: solid; border-color: #000; border-width: 1px">  
    ...  
</div>
```

Interestingly, most major browsers (Safari, Chrome, and Firefox) return shorthand values (despite what the JavaDoc says) for `border`, so the following test passes for them:

[VerifyingStylesIT.java](#)

```
@Test  
public void cssShortHand() throws Exception {  
    driver.get("http://localhost:8080/styled-elements.html");  
  
    WebElement div = driver.findElement(By.id("shorthand"));  
  
    assertEquals("1px solid rgb(0, 0, 0)", div.getCssValue("border"));  
}
```

We recommend avoiding shorthand. For `background`, what you get depends on the browser. The following code works on Safari and Chrome, but not on Firefox:

[CssShorthandIT.java](#)

```
@Test
public void cssShortHandBackground() throws Exception {
    driver.get("http://localhost:8080/styled-elements.html");

    WebElement div = driver.findElement(By.id("shorthand"));

    assertEquals(
        "rgba(0, 0, 0) none repeat scroll 0% 0% / auto padding-box border-box",
        div.getCssValue("background")
    );
}
```

You can see that the `background` shorthand has many parts. When you need to check CSS values, don't check short-hand values. Rather than checking short-hand values, check each value separately:

```
WebElement div = driver.findElement(By.id("shorthand"));

assertEquals("rgb(0, 0, 0)", div.getCssValue("border-color"));
assertEquals("solid", div.getCssValue("border-style"));
assertEquals("1px", div.getCssValue("border-width"));
```

Examining specific elements

There are a bag of different ways to check other part of your page. In this section we'll go through some of the ones we've not talked about before so you have seen a (hopefully) complete set of them.

Examining elements using `getAttribute`

Form elements are a special case. If you want to verify them, you need to use `getAttribute` to examine most form input. In this section, you'll verify the form in figure [Pre-filled Form](#).

Pre-filled Form

Email

john.doe@swip.com

Password

.....

How did you hear about us?

Friend



Contact me by email. Contact me by phone.

Please choose what you are interested in

Books

Music

Movies

I accept the terms and conditions

Comments

Tell us what you think.

Sign-up

Figure 3. Pre-filled Form

To check a field's value, you can get the `value` attribute:

```
assertEquals("john.doe@swb.com",
    driver.findElement(By.name("email")).getAttribute("value"));
```

Password input

You may have thought it be impossible to get the value of a password, because you might imagine it's a security issue. But fear not: you use the same techniques as for text inputs:

```
assertEquals("secret",
    driver.findElement(By.name("password")).getAttribute("value"));
```

TextArea

Text areas don't have a `value` attribute. Instead, you examine their text:

```
assertEquals("Tell us what you think.",
    driver.findElement(By.name("comments")).getText());
```

Check boxes and radio buttons

Check boxes and radio buttons can be verified using the `WebElement.isSelected` method:

```
assertTrue(driver.findElement(InputBy.label("phone")).isSelected());
```

Select drop-downs

Just as when you interact with a select drop-down, you wrap it using the `Select` class. If you're only checking one option, then you can use the `getFirstSelectedOption` method:

```
Select hearAboutSelect = new Select(driver.findElement(By.name("hearAbout")));
assertEquals("Friend", hearAboutSelect.getFirstSelectedOption().getText());
```

If the select has multiple options, you can use `getAllSelectedOptions`, but you'll want to extract the selected text:

[VerifyingFormIT.java](#)

```
WebElement interestsElement = driver.findElement(By.name("interest"));
List<String> selectedText = new ArrayList<>();

for (WebElement option : new Select(interestsElement).getAllSelectedOptions()) {
    selectedText.add(option.getText());
}

Collections.sort(selectedText); (1)

assertEquals(Arrays.asList("Movies", "Music"), selectedText);
```

1. This sort makes the following assert stable, even if the element order changes.

Summary

- WebDriver provides most of the methods you need in the `WebElement` class. The main ones are `isDisplayed` , `getText` , `getCssValue` , `getAttribute` , and `isSelected` .
- You cannot find the text of an `input` using the `getText` method, use `getAttribute("value")` instead.`
- Use the `Select` class to assist verifying select boxes.
- Be careful about checking a page based on its page source. Different browser can return different values
- CSS shorthand can be different across browsers, but you might want to prefer the longhand values as this is likely to be more reliable.

Now that you've seen how to examine a page, in the next chapter we'll look at extracting common automation code using the Page Object pattern. This will make completing forms easier by abstracting away the details of checking boxes and choosing options from select lists.

1. https://en.wikipedia.org/wiki/Behavior-driven_development

Chapter 4: Making maintainable tests using the Page Object pattern

The chapter covers

- Using the Page Object pattern
- Understanding best practices for page objects
- Working with classes for creating page objects

The *Page Object pattern* is almost certainly the best known pattern when it comes to page automation. There's a good reason for this—it's incredibly effective at reducing the duplication and complexity, and increasing reusability and robustness, of automation code. This is especially true as your application grows.

A page object wraps up a page, or part of a page, into a single object, and that object can be reused in multiple places. For example, you might want to wrap a login form or page navigation into a page object. That way, you can reuse that object, rather than having to repeat yourself. If the login form or navigation changes, you only need to change your code in one place—the page object.

In this chapter, we'll visit the fundamentals of page objects, look at some examples of their use, and discuss what makes a good page object and what makes a bad one. Finally we'll look at provided libraries that can help reduce the complexity of your page objects.

What is a page object?

Before we start with techniques and best practices, let's look at the fundamentals.

A page object is an object that encapsulates the behavior of a part, or whole, of a single page. It abstracts away the details of interacting with the HTML of a page, and replaces it with an API that talks in terms of the functionality the pages provides.

For example, a login form might have several form inputs and buttons. To log in using WebDriver, you'd need to set the value of the username and password, and then click the "Login" button. If you use a page object, you might instead only need to ask the

object to "login". If the login form changes, the page object's implementation might need to change, but its API won't. This means that any code that uses the object doesn't need to change.

There are several benefits of using page objects:

- By refactoring code into classes that represent pages, the code that uses these classes can be much easier to understand.
- By reducing duplication, page objects can make code cheaper to maintain.
- Because the concept of a page object is well understood, they can make your code easier for others to work with.

Creating a page object for a login form

Let's consider a login form (see figure [Login form - http://localhost:8080/login.html](http://localhost:8080/login.html)).

The screenshot shows a simple login interface. At the top, the word "Login" is displayed in a large, bold, dark font. Below it is a horizontal row of three input fields. The first field is labeled "Email" and is currently empty. The second field is labeled "Password" and is also empty. To the right of these two fields is a blue rectangular button with the word "Login" in white. To the right of the "Login" button is a blue link that says "Forgotten Password".

Figure 1. Login form - <http://localhost:8080/login.html>

There are a few elements on this form:

- An email input
- A password input
- A submit button
- A "forgotten password" link

The actual HTML looks like this:

[login.html - http://localhost:8080/login.html](http://localhost:8080/login.html)

```
<form class="form-inline" id="login">
  <div class="form-group">
    <input type="text" name="email" class="form-control" placeholder="Email">
    <input type="password" name="password" class="form-control" placeholder="Password">
    <input type="submit" value="Login" class="btn btn-primary">
  </div>
  <div class="form-group forgotten-password">
    <a href="#" id="change-password" class="btn">Forgotten Password</a>
  </div>
</form>
```

Consider how this HTML might translate into an object that encapsulates the behavior of the HTML. Ask yourself – what is the user trying to do when they interact with this part of the page? You might want an API to the form that looks like this:

```
public class LoginForm {
  public void loginAs(String email, String password) {...}
  public void openForgottenPasswordPage() {...}
}
```

Page objects are usually constructed from a single `WebElement` or from the `WebDriver` itself (both of which implement the `SearchContext` interface). This allows the object to find the elements it needs. Using a `WebElement` narrows down the scope of the search and makes sure that you don't accidentally include elements that aren't part of your form.

For a login form, a great choice would be the HTML form element itself, as it contains only elements related to login. By reducing the scope of the object, you reduce the chance that something else on the page changing can impact your object. For example:

```
LoginForm loginForm = new LoginForm(driver.findElement(By.id("login")));
loginForm.loginAs("foo@bar.com", "secret");
```

Our final login form object might look like this:

[LoginForm.java](#)

```

public class LoginForm {
    private final WebElement loginForm;

    public LoginForm(WebElement loginForm) {
        this.loginForm = loginForm;
    }

    public void loginAs(String email, String password) {
        loginForm.findElement(By.cssSelector("input[name='email']"))
            .sendKeys(email);
        loginForm.findElement(By.cssSelector("input[name='password']"))
            .sendKeys(password);
        loginForm.findElement(By.cssSelector("input[type='submit']"))
            .click();
    }
}

```

This object is "lazy". It doesn't search for the input elements when it's constructed. Instead, it searches for them when the `login` method is called. We'll come back to why you might prefer to be "lazy" later on in this chapter.

The page objects in our examples are either constructed from a `WebElement`, or from the `WebDriver`. Both these classes implement the `SearchContext`. All three can be used, but when should you use one and when should you use another?

- `WebDriver` —Use this when you think there will only be one instance of that page object per page. It can also encapsulate the locating of the element into a single place. For example, a navigation menu might be a good choice because you'll only have one of these on a page.
- `WebElement` —Use this when there may be many items on a page, or when the locator you need might change depending on the page you're looking at. For example, a single product in a product listing page (such as on Amazon or eBay) would be a good example, as you may have many of these on a single page.
- `SearchContext` —This can be used in either situation, but it shouldn't be used if the object should only be passed an element.

What might be a good part of a page to make into page objects? Here's a set of examples:

- The navigation menu
- A calendar or date-picker

- Thumbnail images
- A product listing on a shopping site
- The shopping basket
- Almost any form
 - A user-registration form
 - A login form
 - A payment-method registration form, such as for a credit card registration
 - A payment form, such as for a card deposit
- Tables
- An article on a news website or blog
- A search result on a search engine, such as Google
- A user's profile on a social network

Fluent page objects using method chaining

It's said that there's more than one way to skin a cat. One alternative to encapsulating the behavior in a single method is to make the login form a *fluent interface* [1]. The use of fluent interfaces typically starts with an initial initiation step, followed by one or more construction steps, followed by a final execution step. In our example, it would look something like this:

1. Create the page object.
2. Set the username.
3. Set the password.
4. Submit the form.

These steps might result the following code.

[fluentbuilder/LoginFormIT.java](#)

```
new LoginForm(driver.findElement(By.id("login")))
    .username("foo@bar.com")
    .password("secret")
    .submit();
```

This is easier to read for many people than the first version of the login form we looked at. The primary downside to this approach is that although it makes for more readable tests, it makes for more verbose page objects:

[fluentbuilder/LoginForm.java](#)

```
public class LoginForm {
    private final WebElement loginForm;

    public LoginForm(WebElement loginForm) {
        this.loginForm = loginForm;
    }

    public void submit() {
        loginForm.findElement(By.cssSelector("input[type='submit']"))
            .click();
    }

    public LoginForm username(String email) {
        loginForm.findElement(By.cssSelector("input[name='email']"))
            .sendKeys(email);
        return this;
    }

    public LoginForm password(String password) {
        loginForm.findElement(By.cssSelector("input[name='password']"))
            .sendKeys(password);
        return this;
    }
}
```

Creating a page object for a whole page

A page object can represent the whole, or just a part, of a page. This makes a lot of sense if you think about all the parts of a page that might be common within a website:

- Navigation header

- Login form
- Sidebar
- Basket

Each of these can be modeled as a single object, which you can then compose together into an object that represents the whole page. This allows you to reuse those objects in multiple places.

Designing a good generic page object can be a challenge. For example, does the nav bar really appear on every page of the website? Presumably the login form is replaced by a logout form once you’re logged in? Does the basket (or shopping cart) appear on the account pages as well as the pages for browsing products?

An alternative approach to generic page objects comes from applying the *You Aren’t Gonna Need It* (YAGNI) principle [\[2\]](#). YAGNI says, in a nutshell, that you only do work when you know for sure you’ll need it. Rather than have a generic base class that every page object extends, you should instead focus on automating just the part of the page you’re interested in.

What makes a great page object?

Some page objects are badly behaved. They are hard to work with, don’t fully encapsulate the abstraction, require constant attention and create great costs. Other page objects are well behaved, easy to work with and inexpensive to maintain, and bring value to your team. To get the best out of page objects, you can follow some best practices. Let’s look at some of them.

Don’t expose WebDriver methods via public methods or fields

This should probably go without saying. Code that uses a good page object shouldn’t need to be changed if the HTML that page object encapsulates changes—its public API should be stable. This can’t happen if the object exposes its internal working.

The login example is a good demonstration of this. None of its methods expose `WebElement` or `WebDriver` as either a method parameter or return type. Instead, they represent the operations you might want to achieve with the form—logging in.

Model behavior rather than the underlying HTML

Page objects that closely follow the underlying HTML tend to require more lines of code to use. Take a look at the search form in figure [Search form](#):



Figure 2. Search form

You could model a search form as a page object by sticking closely to the form's HTML structure:

a/SearchForm.java

```
public class SearchForm {  
    private final WebDriver driver;  
  
    public SearchForm(WebDriver driver) {  
        this.driver = driver;  
    }  
  
    public void setQuery(String query) {  
        driver.findElement(By.cssSelector("input[name='q']"))  
            .sendKeys(query);  
    }  
  
    public void submit() {  
        driver.findElement(By.cssSelector("input[type='submit']"))  
            .click();  
    }  
}
```

This has a somewhat verbose usage:

```
SearchForm searchPage = new SearchForm(driver);  
  
searchPage.setQuery("funny cats");  
searchPage.submit();
```

But what are you trying to do? Search for something. You could create a class focused on searching:

```
public class SearchPage {  
    private final WebDriver driver;  
  
    public SearchPage(WebDriver driver) {  
        this.driver = driver;  
    }  
  
    public void searchFor(String query) {  
        driver.findElement(By.cssSelector("input[name='q']"))  
            .sendKeys(query);  
        driver.findElement(By.cssSelector("input[type='submit']"))  
            .click();  
    }  
}
```

The usage here is straightforward:

a/SearchFormIT.java

```
SearchPage searchPage = new SearchPage(driver);  
  
searchPage.searchFor("funny cats");
```

The key takeaway here is that you shouldn't be constrained by the HTML. Think about what the HTML helps the user to achieve, and create a page object with an API that matches that.

Be highly cohesive

Imagine what would happen if you created a page object that represented two parts of a page.

```
public void TopBar {  
    ...  
    public void loginAs(String username, String password) {...}  
    public void searchFor(String searchQuery) {...}  
    public void openHomePage() {...}  
}
```

If either the login form, the search page, or the home page changes, you'll need to change this object. Any code that uses this page object could be affected, resulting in more work to maintain your tests. If an object models both the login form and the

search form, any changes to the login form will risk impacting the search form, and vice versa.

Instead, model only one aspect of a page's functionality in a page object. A cohesive object reduces the object's complexity and increases its reusability.

Be lazy

I don't mean you should go and have a nap! I'm talking about *lazy loading* [3].

You may be encouraged to create immutable objects—functional programming is all the rage, and immutable objects are one of its cornerstones. In Java, you can create an immutable object by creating it in a fixed state when you construct it. But here we're modeling something mutable, a web page, which may change.

Consider this page object:

`c/SearchPage.java`

```
public class SearchPage {
    private final WebElement queryInput;
    private final WebElement submitInput;

    public SearchPage(WebDriver driver) {
        queryInput = driver.findElement(By.cssSelector("input[name='q']"));
        submitInput = driver.findElement(By.cssSelector("input[type='submit']"));
    }

    public void searchFor(String query) {
        queryInput.sendKeys(query); (1)
        submitInput.click();
    }
}
```

1. By this time, `queryInput` may already be modified by JavaScript and causes a `StaleElementReferenceException`

The page may not have loaded the HTML that makes up the search when the object is constructed, so the elements might not be accessible. More than that, each time you create this object you make calls to `WebDriver` that you don't need. Doing this unnecessarily will slow down your tests. Also it increases the chance of getting an intermittent `StaleElementReferenceException` during runtime.

Instead, try to find the elements when you need them:

```
public class SearchPage {  
    private final WebDriver driver;  
  
    public SearchPage(WebDriver driver) {  
        this.driver = driver;  
    }  
  
    public void searchFor(String query) {  
        driver.findElement(By.cssSelector("input[name='q']"))  
            .sendKeys(query);  
        driver.findElement(By.cssSelector("input[type='submit']"))  
            .click();  
    }  
}
```

This can dramatically reduce the chance of getting a `StaleElementReferenceException`.

Stale Element Reference Exception

This is commonly encountered exception so Selenium WebDriver website has a detailed document on this exception, please take a look at the following url.

http://docs.seleniumhq.org/exceptions/stale_element_reference.jsp

We will cover this exception in chapter 6.

Throw an error for the wrong page

Typically, it's bad practice to create any object with an invalid state. This applies to page objects just as much as to any other objects. If you create a page object from the wrong page, you might find that your test fails, but when it fails it's hard to diagnose. It might look like you've created a working and valid page object, but that object may not be representing the HTML it should.

When creating a page object, you might want to check that you've got the correct page. Making sure the correct page is loaded, ideally with an inexpensive test, can prevent hard-to-diagnose problems in your code.

[d/SearchPage.java](#)

```

public SearchPage(WebDriver driver) {
    String pageTitle = driver.getTitle();
    if (!pageTitle.equals("Search")) {
        throw new IllegalArgumentException(String.format("page is not search page, it has un-
expected title %s", pageTitle)); (1)
    }
    // ...
}

```

1. You might want to include some diagnostics—if it's not the search page, how will you know what page it actually is?

Don't provide methods that assert

It might be tempting to take your page object and add convenience methods that throw assertion errors under certain conditions.

```

public void assertPageTitleIs(String expectedTitle) {
    assertEquals(expectedTitle, driver.getTitle());
}

```

This couples the page object with your testing framework. You'll typically start to find it hard to change the page objects, because tests are relying on these methods.

Why wouldn't you want to couple your page objects with your testing framework? After all, you're using them in your tests.

- There is more than one framework that a project may use. If you want to share page objects between two projects, you might find that one uses JUnit and the other TestNG, and the page object needs to be changed to be used.
- An assertion is a statement a test makes about the behavior of the application it's testing. A page object is intended to represent the page. Excluding assertions from page objects makes it clear which code is responsible for which.

Instead, you can provide a method that achieves the same thing without the assertion:

```

public void verifyPageTitleIs(String expectedTitle) {
    String actualTitle = driver.getTitle();
    if (!expectedTitle.equals(actualTitle)) {
        throw new IllegalStateException("expected " + expectedTitle + " but got " + actual
Title);
    }
}

```

Using PageFactory and annotations to simplify your page objects

The `PageFactory` class is provided by WebDriver to make writing page objects less verbose and more robust. Let's look at our `SearchForm` again:

```

public class SearchForm {
    private final WebDriver driver;

    public SearchForm(WebDriver driver) {
        this.driver = driver;
    }

    public void searchFor(String query) {
        driver.findElement(By.cssSelector("input[name='q']"))
            .sendKeys(query);
        driver.findElement(By.cssSelector("input[type='submit']"))
            .click();
    }
}

```

The `searchFor` method is doing more things than you might like it to. Specifically, it contains code to identify the elements when you want to use them. What if you add this new method:

```

public void clearQuery() {
    driver.findElement(By.cssSelector("input[name='q']")).clear();
}

```

If you're anything like us, you'll immediately notice some code duplication, and you'll probably want to refactor it out to a method like this:

```

public void searchFor(String query) {
    findQuery()
        .sendKeys(query);
    driver.findElement(By.cssSelector("input[type='submit']"))
        .click();
}

private WebElement findQuery() {
    return driver.findElement(By.cssSelector("input[name='q']"));
}

public void clearQuery() {
    findQuery().clear();
}

```

You may decide to extract the locator into a constant:

[pagefactory/whybad/SearchForm.java](#)

```

public class SearchForm {
    private static final By QUERY_SELECTOR = By.cssSelector("input[name='q']");
    private final WebDriver driver;

    public SearchForm(WebDriver driver) {
        this.driver = driver;
    }

    public void searchFor(String query) {
        driver.findElement(QUERY_SELECTOR)
            .sendKeys(query);
        driver.findElement(By.cssSelector("input[type='submit']"))
            .click();
    }

    public void clearQuery() {
        driver.findElement(QUERY_SELECTOR).clear();
    }
}

```

Another solution would be to have a field containing the query input, but you'd lose the benefits of lazy loading for the element. There's no ideal solution ... or is there?

Using PageFactory to simplify code

The locating of elements in page objects is done with explicit calls to `WebDriver`, and it results in verbose code.

Use `PageFactory` to create page objects with fields that are populated automatically.

`PageFactory` is provided as part of WebDriver's support library.

So that it doesn't look like magic, let's talk a bit about how `PageFactory` works.

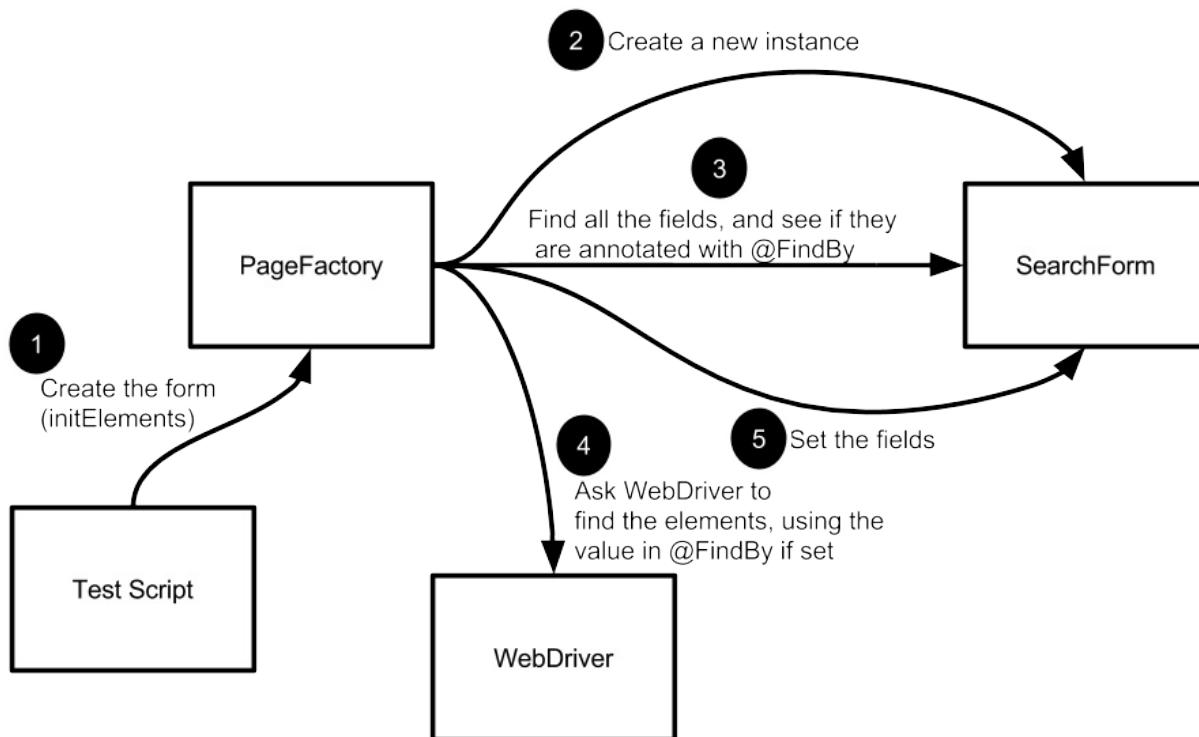


Figure 3. Page Factory

Firstly, you'll call `PageFactory.initElements(driver, YourPageObject.class)`. This creates an instance of your object.

Page objects that use `PageFactory` must be created using `initElements`. If this is not used, the object will have null fields and will not work as expected.

Then, for each field in your object that is a `WebElement`, `PageFactory` determines the correct locator to find it. If it is annotated with the `@FindBy` annotation, then the `PageFactory` will use that to determine the locator. Otherwise, it assumes there is an element on the page whose ID, or name is the same as the field's name.

Lets have a look at our search form:

<http://localhost:8080/search.html>

```
<form class="form-inline" role="search">
  <div class="form-group">
    <input type="text" name="q" class="form-control" placeholder="Search"/>
    <input type="submit" value="Search" class="btn btn-primary"/>
  </div>
</form>
```

For this form, we'll want to find the query input, and the submit input. The query input could be located by its name `q`. As we know that if the name of the field matches either the ID or name of the element, then we can use that. So lets start with a basic page object:

```
public class SearchForm {
  private WebElement q;
}
```

The submit input does not have an ID or name, but we could use its type attribute: `submit`. This can be done using a CSS locator, so we can add a `@FindBy` annotation with it `css` property set to `input[type='submit']`:

```
public class SearchForm {
  private WebElement q;
  @FindBy(css = "input[type='submit']")
  private WebElement submit;
}
```

Next, lets add a `searchFor` method:

```
public class SearchForm {
  private WebElement q;
  @FindBy(css = "input[type='submit']")
  private WebElement submit;

  public void searchFor(String query) {
    this.q.sendKeys(query);
    this.submit.click();
  }
}
```

Finally, I'd like to rename the variable `q` to something more descriptive. I'd like to rename it to `query`. But the input is not named `query`, so I'll need to add a `@FindBy` annotation with its `name` property set:

[pagefactory/SearchForm.java](#)

```
public class SearchForm {
    @FindBy(name = "q")
    private WebElement query;
    @FindBy(css = "input[type='submit']")
    private WebElement submit;

    public void searchFor(String query) {
        this.query.sendKeys(query);
        this.submit.click();
    }
}
```

To create this page object, you must invoke the ``PageFactory.initElements`` method:

[pagefactory/SearchFormIT.java](#)

```
SearchForm searchForm = PageFactory.initElements(driver, SearchForm.class);
searchForm.searchFor("funny cats");
```

When you create a page object in this way, `PageFactory` populates each of the fields. One of the best practices we talked about earlier was lazy loading, and this might seem to be opposite of this-eager loading, but there's some magic!

For each of the fields, instead of a real, concrete element, `PageFactory` creates a *dynamic proxy* [4]. The proxy holds a reference to the original driver instance the object was created from, and the locator it decided to use. Each time a method is called on the proxy, it locates the element afresh, making sure you can't have a stale element.

Creating elements lazily is great for most cases, because every invocation makes sure the element is available. But if your code is slow, and you know the element is always available on the page, and you are using it repeatedly, then annotate the field with the `@CacheLookup` annotation. The element will only be looked up once.

`PageFactory` can help reduce the verbosity of code by changing how you define a page object. Instead of explicitly getting elements when you use them, `PageFactory` will populate them dynamically based on the locators you've configured them with.

This is great for most cases, but you should be aware of a couple of things.

First, as mentioned before, the fields are populated with proxies rather than concrete elements. This can be seen in figure [Debugging a page object created by `PageFactory`](#), where the fields are of type `com.sun.proxy.$Proxy` rather than `WebElement`.

```
▼ └─ searchForm = {swip.ch06pageobjects.pagefactory.v1.SearchForm@2307}
  └─ └─ query = {com.sun.proxy.$Proxy20@2319} "<input type="text" name="q" class="form-control" placeholder="Search" />"
    └─ └─ h = {org.openqa.selenium.support.pagefactory.internal.LocatingElementHandler@2365}
      └─ └─ submit = {com.sun.proxy.$Proxy20@2320} "<input type="submit" value="Search" class="btn btn-primary" />"
```

Figure 4. Debugging a page object created by `PageFactory`

This means that even if you know the type of the element produced by a specific driver, the actual object you get might not be an instance of the class you expect. You shouldn't write code that makes this assumption.

Second, it's possible to create an instance of a page object using the standard `new` command. This object will not have its fields set—they will be null. Finally, because `PageFactory` requires a public constructor, it's not possible to make the object constructor private.

Using LoadableComponent to make your page objects robust

When writing automation code, you are likely to find yourself doing the following on a regular basis:

1. Execute an operation that you expect to load a page.
2. Verify the page loaded correctly.
3. Create a page object for the loaded page.

Using LoadableComponent to make your page object robust

Page objects are loaded, but you don't know if they've loaded from the correct page. This results in tests that may fail unexpectedly.

Use `LoadableComponent`.

`LoadableComponent` is provided as part of WebDriver's support library, and its goal is to support the writing of page objects and reduce the amount of code you need to write.

To utilize it, you need to provide two pieces of code on your page object:

1. Code to load the page.
2. Code to verify that the page is loaded.

`LoadableComponent` then uses those two blocks of code to load and then verify the page in a single step.

Let's redo our search page in this style:

[loadable/SearchForm.java](#)

```
public class SearchPage extends LoadableComponent<SearchPage> {  
    private final WebDriver driver;  
  
    public LoadableSearchPage(WebDriver driver) {  
        this.driver = driver;  
    }  
    @Override  
    protected void load() {  
        driver.get("/search.html");  
    }  
  
    @Override  
    protected void isLoaded() throws Error {  
        assertEquals("Search", driver.getTitle()); (1)  
    }  
  
    public void searchFor(String query) {  
        driver.findElement(By.cssSelector("input[name='q']")).sendKeys(query);  
        driver.findElement(By.cssSelector("input[type='submit']")).click();  
    }  
}
```

1. This is a violation of early advice – don't use assert in a page object – but this is an exception as this is how this class is designed to be used.

The usage is straightforward:

[loadable/SearchFormIT.java](#)

```
LoadableSearchPage page = new LoadableSearchPage(driver).get();
page.searchFor("funny cats");
```

There are unattractive issues with using `LoadableComponent`:

- You can create an unloaded version of the page, effectively creating an object in an invalid state.
- It encapsulates the logic within methods that can be overridden. If there are many subclasses, it makes them hard to reason about.
- The contract of the method `isLoading` states you must throw an `Error` "when the page is not loaded". In Java, an `Error` "indicates serious problems that a reasonable application should not try to catch". This is not (IMHO) the most appropriate exception. Perhaps returning a boolean would be preferable?

What we're trying to do with `LoadableComponent` is an excellent idea. In the next section, we'll look at bringing `LoadableComponent` and `PageFactory` together, so we can get the benefits of both.

Creating a loading page factory

We'd like to get the benefits of loadable components and page object factories at the same time. This technique shows how to combine the two.

We'd like a unified approach to loading, populating, and verifying page objects.

Create a `LoadingPageFactory` class to do these operations.

Let's quickly review the benefits of page objects, `PageFactory`, and `LoadableComponent` and create some requirements for our new factory:

- Encapsulating page logic and enabling reuse
- Simplifying the definition of a page object
- Enabling lazy loading of elements
- Preventing the creation of invalid page objects

To do this we need three things:

- A strategy for loading the page
- A strategy for verifying that the page has loaded

- A strategy for locating elements

You can do this all using annotations. We're going to create some new annotations:

`@Path` and `@Verify` to work with `@FindBy`. Lets look at how we would use them:

[loadablepagefactory/SearchForm.java](#)

```
@Path("http://localhost:8080/search.html")
@Verify(title = "Search")
public class SearchForm {

    @FindBy(css = "input[name='q']")
    private WebElement query;
    @FindBy(css = "input[type='submit']")
    private WebElement submit;

    public void searchFor(String text) {
        query.sendKeys(text);
        submit.click();
    }
}
```

We want a terse way to load these pages, so borrowing from `PageFactory` gives us the following:

[loadablepagefactory/SearchFormIT.java](#)

```
SearchForm page = LoadingPageFactory.get(driver, SearchForm.class);
```

This can be implemented in a few lines of code, as follows.

[LoadingPageFactory.java](#)

```

public class LoadingPageFactory {
    public static <T> T get(WebDriver driver, Class<T> pageObjectClass) {

        driver.get(pageObjectClass.getAnnotation(Path.class).value()); (1)

        Verify verify = pageObjectClass.getAnnotation(Verify.class); (2)

        String expectedPageTitle = verify.title();
        if (!expectedPageTitle.equals(Verify.INVALID_TITLE)) { (3)
            String actualPageTitle = driver.getTitle();
            if (!expectedPageTitle.equals(actualPageTitle)) {
                throw new IllegalStateException(
                    String.format(
                        "expected page title %s but was %s",
                        expectedPageTitle,
                        actualPageTitle
                    )
                );
            }
        }

        return PageFactory.initElements(driver, pageObjectClass); (4)
    }
}

```

1. Gets the page
2. Assumes that the @Verify annotation will be present
3. Verifies the page title if it's defined
4. Delegates the populating of elements to PageFactory

Finally, you need the two new annotations:

[Path.java](#)

```

@Retention(RetentionPolicy.RUNTIME) (1)
@Target(ElementType.TYPE) (2)
public @interface Path {
    String value();
}

```

1. Indicates this annotation needs to be available at runtime
2. Indicates this annotation applies to types only

[Verify.java](#)

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Verify {
    String INVALID_TITLE = "\0"; (1)

    String title() default INVALID_TITLE;
}

```

1. This is an invalid page title. This allows the verification of the title to be optional.

This is one way to combine the benefits of `PageFactory` and `LoadableComponent` to consistently create verified page objects. It's a bit light on features, as it can only verify by page title, and other features will be useful in many cases.

Annotations have some caveats. For example, values can't be null, so if an attribute is optional, you'll need to use some illegal value that no client would be expected to pass. Here we use the null string. You also must annotate them with

`@Retention(RetentionPolicy.RUNTIME)` to make sure the JVM makes them available when your code runs, and `@Target(ElementType.TYPE)` to ensure that they are only applicable to class.

You might find this sort of class useful in your automation framework.

Because verifying the title is optional, you could extend this approach to verify other items on the page. For example, you could add a check to see that an XPath exists:

```

public @interface Verify {
    ...

    String INVALID_XPATH = "\0";
    String xpath() default INVALID_XPATH;
}

```

To implement this, add the following to your `LoadingPageFactory`:

>LoadingPageFactory.java

```

String xpath = verify.xpath();
if (!expectedPageTitle.equals(Verify.INVALID_XPATH)) {
    if (driver.findElements(By.xpath(xpath)).isEmpty()) {
        throw new IllegalStateException(String.format("expected XPath %s", xpath));
    }
}

```

This will allow you to check for a part of a page:

```
@Verify(xpath = "//h1[text()='Search']")
public class SearchForm {
    ...
}
```

Summary

- Page objects can represent the whole of a page, or just part of a page. You can choose the best approach depending on circumstances.
- Page objects help reduce the complexity of tests, and help make them easier to understand by allowing you to re-use code that works with parts of a page used in many places. Page objects make your tests cheaper to maintain.
- There are a number of best practices for getting the most out of page objects. These include hiding away WebDriver, being cohesive, and lazy loading.
- Selenium's support library provides `PageFactory` and `LoadableComponent`. These help you keep your code short and easy to understand. You can build on these to create your own utility classes.

We don't live in an ideal world, and there are number of issues you'll encounter when using WebDriver for which solutions and work-arounds will be required. In the next chapter we'll look at what you can do when things go wrong.

1. See the "Fluent Interface" article on Wikipedia:

https://en.wikipedia.org/wiki/Fluent_interface

2. "You aren't gonna need it" on Wikipedia:

https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it

3. See the "Lazy loading" Wikipedia article:

https://en.wikipedia.org/wiki/Lazy_loading

4. See "Dynamic Proxy Classes" in Oracle's Java documentation:

<https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html>

Chapter 6: What To Do When Something Goes Wrong

In this chapter:

- Retrying when you can't find a slippery element
- Understanding implicit waits
- Explicit waiting
- Looking for alternative locators
- Internationalization

If only everything in life worked well! But that's not always the case. Your tests have to cooperate with WebDriver, with the browser, and with the web site you are testing. You don't want a set of tests that are failing most of the time due to issues with the tests, rather than issues with the application you are testing. These false positives may mean that you or your team start to see "red" as an indicator of a problem with the test suite, rather than a problem with your application, and gain a false sense of security. This is the dreaded "flaky test suite", which you'll either rapidly become habituated to and one day miss a critical issue, or spend a significant part of your time fixing. Stabilizing your tests is therefore key to keeping you and your team productive.

Here we'll cover common errors when using WebDriver, such as elements that are stale or missing, as well as the causes and possible solutions. For the most common errors, you will see various strategies to mitigate them.

By the end of the chapter, you will have learned a number of techniques that will help you keep your test suites stable and low maintenance.

Exceptions

There are a large number of different exceptions that WebDriver can produce. They fall into two broad categories: those that are caused by problems with WebDriver (such as being unable to connect to the browser), and those that are caused by code not actually matching the behavior of the site you are testing.

Let's have a look at a few of these exceptions.

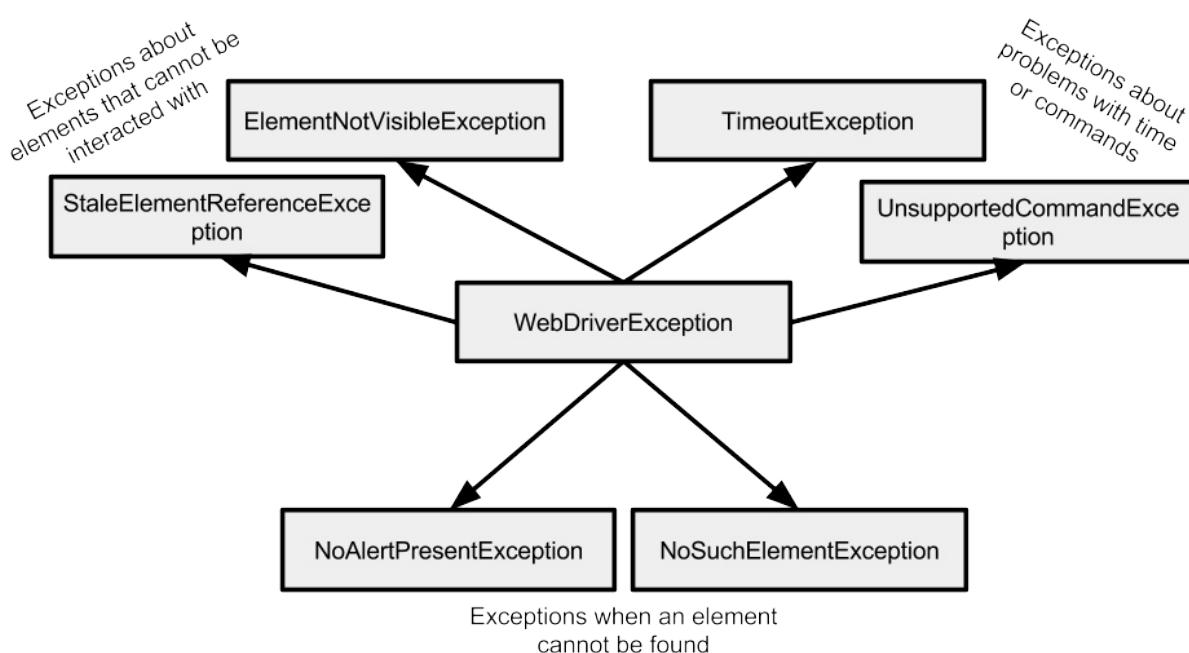


Figure 1. WebDriver Exceptions

ElementNotVisibleException

An element is not visible, and therefore cannot be interacted with.

[CommonExceptionsIT.java](#)

```

@Before
public void setUp() throws Exception {
    driver.get("/webdriver-exceptions.html");
}

@Test(expected = ElementNotVisibleException.class)
public void invisibleElementShouldNotBeVisible() throws Exception {
    driver.findElement(By.id("invisible")).click();
}
  
```

You might want to try and wait for the element to become visible.

NoSuchElementException

This exception is thrown when trying to find an element that cannot be found on the page.

[CommonExceptionsIT.java](#)

```
@Test(expected = NoSuchElementException.class)
public void noSuchElement() throws Exception {
    driver.findElement(By.id("no-such-element!")).click();
}
```

You might want to wait for the element to appear, or you might be on the wrong page.

StaleElementReferenceException

This exception occurs when trying to interact with an element that is no longer on the page, for example if the element has been removed from the page, or the browser has navigated to a new page.

[CommonExceptionsIT.java](#)

```
@Test(expected = StaleElementReferenceException.class)
public void elementShouldBeStaleWhenPageChanges() throws Exception {
    WebElement button = driver.findElement(By.id("button"));
    driver.get("/");
    button.click(); (1)
}
```

1. This button is no longer visible.

You need to try and locate the element again.

TimeoutException

This exception is thrown when a WebDriver has been waiting for an operation to complete, but time ran out. If you're waiting for an element to appear, and it has not within the timeout, then the web application might have changed.

NoAlertPresentException

This exception is thrown when trying to interact with a JavaScript alert, but this alert does not exist.

UnsupportedCommandException

Indicates that an operation you requested is not supported by the browser you are using.

`WebDriverException` is the root of all WebDriver exceptions. Catching this catches all exceptions that WebDriver might produce. WebDriver produces superbly detailed and useful exceptions. Let's look at some of the information that you can find out from them:

WebDriver build information, telling you how old your version is (e.g. if it is time to upgrade)

"Build info: version: '2.52.0', revision:

'5017cb8e7ca8e37638dc3091b2440b90a1d8686f', time: '2015-02-27 09:10:26'"

System information, so you can track problems to the OS or Java version

e.g. "System info: host: 'GL04321M.lan', ip: '192.168.1.82', os.name: 'Mac OS X', os.arch: 'x86_64', os.version: '10.10.3', java.version: '1.8.0_40'"

Support URL with more information about the problem

e.g. "http://seleniumhq.org/exceptions/no_such_element.html"

Information about the driver's capabilities (such as the browser's name, if it supports JavaScript, or can take a screenshot)

e.g. "Driver info: org.openqa.selenium.firefox.FirefoxDriver Capabilities
[{applicationCacheEnabled=true, rotatable=false, handlesAlerts=true, databaseEnabled=true, version=38.0.5, platform=MAC, nativeEvents=false, acceptSslCerts=true, webStorageEnabled=true, locationContextEnabled=true, browserName=firefox, takesScreenshot=true, javascriptEnabled=true, cssSelectorsEnabled=true}]

The current session

e.g. Session ID: 3c0185ae-b29d-e14d-84e7-d0034e3fa707

Information about the exception

e.g. Element info: {Using=class name, value=does-not-exist}"

If you are using an older driver, and have recently upgraded your browser, you may be able to solve some problems by updating your version of WebDriver to the latest version. Certainly, this can be useful information when investigating issues, or filing bug reports.

Retrying when you can't find a slippery element

One of the most common problems you will encounter when using WebDriver is being unable to find an element. A **slippery element** is one that might have been in the browser once, but is no longer there. For example:

- The browser might have moved to a new page, but you still have a variable pointing to an element on the previous page.
- The element might have been deleted from the page, for example an error message that has been dismissed by the user.
- The element might have become detached from the page's DOM, for example if JavaScript has removed the element but planned to re-use it later on.

Both of the latter are good candidates if the page you are testing uses a lot of JavaScript. The clearest symptom of a slippery element is a

```
StaleElementReferenceException .
```

Retrying on exception

If you have a variable pointing to an element, but that element becomes stale, then calling methods on it will result in a `StaleElementReferenceException`. We can retry locating the element when this happens.

Tests are unreliable due to slippery elements.

Transparently retry the locating of your element:

TransparentRetryIT.java

```
driver.get("/login.html");

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

driver.get("/index.html");

try {
    email.sendKeys("foo@bar.com");
} catch (StaleElementReferenceException ignored) {
    email = driver.findElement(By.name("email"));
    email.sendKeys("foo@bar.com");
}
```

This problem can be caused by code that holds a reference to an element longer than it is available for. The above example is a little contrived, but the principle holds for any page that can remove elements: catch the exception and re-find the element.

If you find yourself doing this regularly, you may want to review your code. It might be caused by logic that could be improved.

Understanding implicit waiting

Many modern web pages use JavaScript and CSS to create animations when you click on elements. In the example application, we have a page with elements that take a few seconds to load [1].

Slow Loading Elements

Fade In The Text

Fade Out The Text

Some slowly loading text.

Figure 2. Slow Loading Elements Page

These animations can be the enemy of automation, as they can mean that your tests will become brittle. This is especially important if the time that elements take to load might vary, or if parts of the page are loaded asynchronously from a remote server.

In Java, if we want to wait for something, we can use the `sleep` method on the `Thread` class. This causes the program to wait (i.e. sleep) for the specified period of time:

```
Thread.sleep(1000); // sleep for 1000ms
```

Unfortunately, it can be hard to figure out the right amount of time to sleep. You might wait longer than you need to, resulting in slow tests, or too little time, resulting in failing tests. The use of `Thread.sleep` is a "test smell" – I've seen over and over again in test suites that are slow and unreliable.

Lets look at how we can avoid sleeping. WebDriver has a built-in mechanism for waiting – implicit wait. The implicit wait is the amount of time that WebDriver should always wait for elements to be ready to use. It can be set as follows:

[ImplicitWaitIT.java](#)

```
driver.manage().timeouts().implicitlyWait(1, TimeUnit.SECONDS);
```

Using implicit waits is considered bad practice:

- You may set it in one test, and find that as it will still be configured for another test, your tests become flaky.
- It can make for slow-running tests, as failing finds (e.g. for invisible or nonexistent elements) will take longer to execute.
- When mixing implicit waits with explicit waits, you'll find waiting times become unpredictable.

If you do use implicit waits, make sure you reset them at the end of your tests:

ImplicitWaitIT.java

```
@After  
public void resetImplicitWait() {  
    driver.manage().timeouts().implicitlyWait(0, TimeUnit.SECONDS);  
}
```

Explicitly waiting for elements to load

An alternative to implicit waits is explicit waits. In this case, you wait for a condition to be true. This might be a condition that expects an element to be visible, or exist on the page. There are two key classes for explicit waits: `WebDriverWait` and `ExpectedConditions` classes. `WebDriverWait` is a class that actually does the waiting. The `ExpectedConditions` class is a factory that can produce the common conditions you might want to wait for.

Using WebDriverWait to wait for slippery elements

Sometimes we need to wait until a condition becomes true, as we know something will happen, but we're not sure when. This technique will look at using `WebDriverWait` to wait for something to change in the page.

You have a web page with elements that take a few moments to load and you need your test to wait until they have loaded.

Use `WebDriverWait` to poll for a condition and return when it is true. If the condition does not become true within the required time, then a `TimeoutException` is thrown.

The condition can be one of two things:

- A boolean predicate

- An **expected condition**

You'll generally find expected conditions the most useful, as an expected condition can also return an element that you can then use. This is fantastic if you are waiting for an element to become visible, as in the following example:

ExplicitWaitIT.java

```
driver.get("/slow-loading-elements.html");

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

WebDriverWait wait = new WebDriverWait(
    driver,
    3, (1)
    100 (2)
);

final WebElement paraElement = wait
    .withMessage("could not find the slowly loading text") (3)
    .until(
        ExpectedConditions
            .visibilityOfElementLocated(By.id("theText")) (4)
    );

assertEquals("Some slowly loading text.", paraElement.getText());
```

1. How long to wait in seconds, in this case 3 seconds.
2. How often we want to poll for the condition to be true.
3. A message to append to the exception when the condition is not satisfied within the time limit.
4. A condition: the element must be visible.

`WebDriverWait` allows you to wait for any condition, not just a visible element. Let's have a look at some of the useful conditions.

Method Name	Description
<code>alertIsPresent</code>	A JavaScript alert is open.
<code>elementToBeClickable</code>	An element is both available and enabled.
<code>frameToBeAvailableAndSwitchToIt</code>	A frame exists, and also switch to the frame.
<code>invisibilityOfElementLocated</code>	An element is not visible.
<code>presenceOfElementLocated</code>	The element found by a locator is on the page (watch out – it might not be visible).
<code>textToBePresentInElement</code>	An element contains some text - useful for waiting for a warning message.
<code>textToBePresentInElementValue</code>	A form input's value contains a string.
<code>titleContains</code>	The page's title contains a certain value. Useful if your page titles have a common prefix.
<code>titleIs</code>	The page's title becomes a certain value.
<code>visibilityOfElementLocated</code>	The element is visible on the page.

If you need to check two conditions, you can use another wait.

ExplicitWaitIT.java

```
WebElement paraElement = wait
    .until(ExpectedConditions.visibilityOfElementLocated(By.id("theText")));

wait
    .until(ExpectedConditions.textToBePresentInElement(paraElement,
        "Some slowly loading text."));
```

The class `WebDriverWait` is a specialisation of the `FluentWait` class that makes it simpler to use. You can use `FluentWait` anywhere you `WebDriverWait` for example,

ExplicitFluentWaitIT.java

```
FluentWait<WebDriver> wait = new FluentWait<>(driver)
    .withTimeout(3, TimeUnit.SECONDS) (1)
    .pollingEvery(100, TimeUnit.MILLISECONDS) (2)
    .ignoring(NotFoundException.class); (3)

WebElement paraElement = wait
    .withMessage("could not find the slowly loading text")
    .until(
        ExpectedConditions
            .visibilityOfElementLocated(By.id("theText")))
);
```

1. Specify the timeout.
2. Specify the polling interval.
3. Choose which exceptions you want to ignore.

You can see you need more lines of code to use it, but you may find it clearer. So, when would you use one or the other?

Use `WebDriverWait` unless:

- You want the code to be clear as to the timeout and polling intervals.
- You need an unusual timeout. One that is less than a second for example.
- You want to use an unusual polling interval. For example, one that is longer than a second.
- You want to ignore exceptions other than `NotFoundException`, which `WebDriverWait` ignores by default.
- You need to find one element within another using `wait` (we'll show an example shortly).

Using waits often requires some fine-tuning. You want to wait the maximum likely time for an element to meet the condition you are waiting for, but no longer. We will go more in-depth with explicit waiting in Chapter 13.

Explicit Wait

Starting from Chapter 13, we are going to show you a journey of developing a framework based on Selenium WebDriver. We will provide a method with build-in waiting with explicit wait so you don't need to debate whether to choose `WebDriverWait` or `FluentWait`. You no longer need to use them directly, the framework will take care of that for you. We will put that method with wait inside an interface. And we name that interface `ExplicitWait`.

Looking for alternative locators

You may also encounter a problem, and even after you have tried those techniques, you still can't find the element using WebDriver. If that's the case, you can look for alternative locators. Let us have a look of this example.

For example, you want to automate this ReactJS Datepicker from Hackerone. First, you need find the input field, and click it to display the calendar.



Figure 3. ReactJS Datepicker

You use Web Developer Tool to inspect the element and find it has a `class` attribute "`ignore-react-onclickoutside`", but it doesn't have an `id` or `name`.

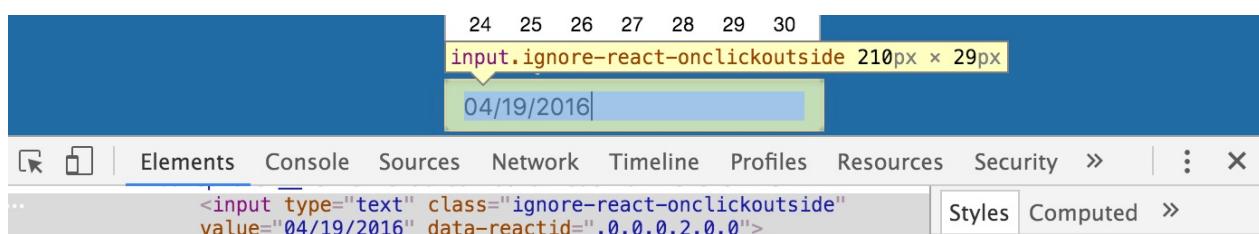


Figure 4. Input field with class attribute

So lets try using the class name:

[FindByClassNameIT.java](#)

```
@Test(expected = NoSuchElementException.class)
public void failedToLocate() {
    browser.findElement(By.className("ignore-react-onclickoutside"));
}
```

It doesn't work, the finder method throws a `NoSuchElementException`. You apply the techniques learned from previous section and add explicit wait to the finders, but it doesn't work either.

FindByClassNameIT.java

```
WebDriverWait wait = new WebDriverWait(driver, 3, 100);
wait.until(new Function<WebDriver, WebElement>() { (1)
    @Override
    public WebElement apply(WebDriver webDriver) {
        return driver.findElement(
            By.className("ignore-react-onclickoutside"));
    }
});
```

1. It will throw a `TimeoutException`

We inspected the page again and find that the input field is inside a container `div` with `class` attribute `"react-datepicker__input-container"`. We learnt from Chapter 2 that we can narrow down the search by searching from a container element. Use its class to find it,

FindByClassNameIT.java

```
WebElement webElement = driver.findElement(
    By.className("react-datepicker__input-container"));
```

We find the container using this approach. But when we try to locate the input field from this container, it fails again.

FindByClassNameIT.java

```
FluentWait<WebElement> webElementFluentWait = new FluentWait<>(webElement)
    .withTimeout(3, SECONDS)
    .pollingEvery(20, MILLISECONDS)
    .ignoring(NoSuchElementException.class);
webElementFluentWait.until(new Function<WebElement, WebElement>() { (1)
    @Override
    public WebElement apply(WebElement webElement) {
        return webElement.findElement(
            By.className("ignore-react-onclickoutside"));
    }
});
```

1. It throws a `TimeoutException`

Even we change the timeout to 30 seconds, we are still not able to find the trigger by its class name. And this datepicker is built from third party library, they may never provide `id` or `name` attribute for us to use. So we need to look for alternative locators.

Find by TagName alternative

Since we can find its container, we can try to find the `input` from the container using `By.ByTagName` locator,

[FindByTagNameIT.java](#)

```
@Test
public void locateSuccessfully() {
    driver.findElement(By.className("react-datepicker__input-container"))
        .findElement(By.tagName("input")).click(); (1)
}
```

1. It doesn't need wait to find the input field.

It works, we can see it triggers the display of the calendar.

Find by Xpath alternative

We know we can find it using xpath,

[FindByXpathIT.java](#)

```
@Test  
public void locateSuccessfully() {  
    driver.findElement(  
        By.xpath("//[@id=\"app\"]/descendant::input")).click();  
}
```

It also works, we can see the calendar is displayed when we run the test.

We use alternatives and find the element we are looking for. Don't try to stick to a single way of doing things.

Datepicker

Datepicker is a complex web widget so we will dedicate two chapters to explain the techniques to automate datepicker. And we will develop a framework to automate 5 kinds of datepickers including the most contemporary JavaScript libraries such as Material-UI and ReactJS in Chapter 17 and 18. Also, we will use the framework developed in Part 3 to improve the tests shown here in Chapter 18.

Internationalization

WebDriver's JSON protocol uses UTF-8 character encoding. However, you need to make sure your source code uses the correct encoding. If you are using Java properties files, the encoding of those files is always ISO-8859-1. But you can use various tools to work around this.

Extracting link text to resource bundles for internationalization

Most public-facing web sites are in multiple languages. However, the functionality is the same in all versions, with text translated (and perhaps some image changes). This technique will look at how you can avoid copying and pasting tests by using resource bundles to store the text in different languages.

You have a multilingual site, and you want to test in different languages.

Extract all text from your code and store it in a resources bundle. Then run the code using a system property to switch languages. Imagine you have a Spanish login page:

Iniciar Sesión

The form consists of four input fields: 'Email' (gray), 'Contraseña' (gray), 'Login' (blue button), and 'Contraseña Olvidada' (blue link). The 'Login' button is highlighted.

Figure 5. Spanish Login Form <http://localhost:8080/es/login.html>

The link text "Forgotten Password" is "Contraseña Olvidada" in Spanish. The above code will not work. We will use Java's resource bundle to store the link text in different languages.

Create, in the root of your code's resources (for Maven projects that is `src/main/resources`), a file named `strings.properties`:

```
forgotten.password=Forgotten Password
```

Create a Spanish file, `strings_es.properties`:

```
forgotten.password=Contraseña Olvidada
```

At this point, it is worth checking that the properties file is in the ISO-8859-1 format, rather than UTF-8. Properties files have to be in ISO-8859-1. You can do this using the `file` command on UNIX, Linux, or Mac, or Cygwin on Windows:

```
% file strings_es.properties
strings_es.properties: UTF-8 Unicode text, with no line terminators
```

Uh oh! It is UTF-8. The "ñ" is not an ISO-8859-1 character, and needs to be replaced by the **escaped unicode character** [2]. You can use `native2ascii` to convert the file:

```
% native2ascii -encoding UTF-8 strings_es.properties
forgotten.password=Contrase\u00f1a Olvidada
```

You can then use the `ResourceBundle` class to get the value:

[ResourceBundleExampleIT.java](#)

```
driver.get("http://localhost:8080/login.html");

ResourceBundle strings = ResourceBundle.getBundle("strings", Locale.ENGLISH);

driver.findElement(By.linkText(strings.getString("forgotten.password")));
```

Or for Spanish:

[ResourceBundleExampleIT.java](#)

```
driver.get("http://localhost:8080/es/login.html");

ResourceBundle strings = ResourceBundle.getBundle("strings",
    Locale.forLanguageTag("es"));

driver.findElement(By.linkText(strings.getString("forgotten.password")));
```

This shows a straightforward technique using core Java libraries to handle internationalization in the code. It is an interesting challenge to develop code that runs under different configurations, such as different browsers and languages. We will discuss this in more detail later in the book.

You set the language the code is running in a system property.

```
mvn verify -Duser.language=es
```

And you can then change your code to:

[ResourceBundleExampleIT.java](#)

```
String language = Locale.getDefault().getLanguage();

driver.get("http://localhost:8080/" + language + "/login.html");

ResourceBundle strings = ResourceBundle.getBundle("strings");

driver.findElement(By.linkText(strings.getString("forgotten.password")));
```

It is worth noting at this point that this is not the only way to locate elements based on the text they contain. We will look at a few techniques using XPaths later on, but before that, let's examine the remaining major locators.

As a final note, the `iconv` tool can also convert text files between encodings.

Summary

- WebDriver has a number of exceptions that it can throw. Some are more common than others, and can be grouped by their type.
- The `Thread.sleep` command will make your test suite slow and unreliable. Try and use a wait instead.
- Implicit waiting has some issues that make for brittle tests. It should be avoided.
- Explicit waits, combined with expected conditions, can resolve page timing issues. `WebDriverWait` provides a number of built in conditions you can wait for.
- Internationalization can be addressed using resource bundles. There are various command line programs that can help with this.

In the next chapter, we'll look at ways of managing WebDriver itself to reduce problems.

1. <http://localhost:8080/slow-loading-elements.html>
2. https://en.wikipedia.org/wiki/List_of_Unicode_characters

Part 2: WebDriver APIs In Depth

In this section we will dive into how to use the WebDriver APIs for automating more complex page elements. We'll also look at automating JavaScript web applications. Finally, we'll introduce you to each of the main drivers, such as the Firefox Driver, as well as mobile drivers for iOS and Android.

Chapter 7: Managing WebDriver

This chapter covers:

- How to make sure browsers are properly set up and cleaned up
- Using **Dependency Injection** to manage your driver

We have all seen it. You've created a fantastic suite of tests. You've set up a Continuous Integration server, such as Jenkins, which is running your test suite regularly and notifying you of any problems with your website. But one day, the Continuous Integration server starts running slowly. You log in to it remotely, only to find dozens of open browsers that have failed to quit. Or perhaps your script has run on another server and failed, but the browser has closed so you do not know why. Or maybe you have a suite of tests that you are running against a test environment, but you need to run them against a preproduction environment, or you have a suite of tests that you are running against one web driver, but you need to run them against other versions. Perhaps your suite is opening a new browser for every test and is extremely slow.

Many of these problems come from "unmanaged" web drivers. But what do I mean by a "managed driver"? It's a driver that is managed by a part of your software. That part is responsible for creating and quitting the driver, so that your tests can focus on testing. In this chapter we will look at how the **Dependency Injection** design pattern can be used to achieve more robust test suites. As usual, we will try to ensure they can be retrofitted to existing code bases.

Dependency Injection in non-Java Languages

In this chapter we focus on using the Java Spring framework. Martin Fowler has an excellent article on his website describing it in detail:

<http://www.martinfowler.com/articles/injection.html>. Frameworks in other languages also exist, and these can also be used to manage drivers.

By the end of this chapter you will have learned:

- How to make sure browsers are closed at the end of your tests
- How to use Dependency Injection to reuse a single web driver
- How to take a screenshot automatically

Quitting the driver, even if the Java Virtual Machine crashes

The Java Virtual Machine (JVM) can crash, or be forcibly terminated. When this happens, the web driver might not have an opportunity to quit and the browser can remain open.

You have probably seen a JUnit test that sets up a driver and then quits in the teardown method, something similar to the following:

CommonWebDriverIT.java

```
public class CommonWebDriverIT {  
  
    private final WebDriver driver = new FirefoxDriver();  
  
    @After  
    public void tearDown() throws Exception {  
        driver.quit();  
    }  
  
    @Test  
    public void checkTheRegistrationPage() throws Exception {  
        // get page, interact and then verify  
    }  
}
```

On initial inspection, everything looks fine. You are creating a driver, using it in your test, and then quitting it when complete. This will work fine to start with, but what happens if you kill the JVM mid-test? The browser will stay open. JUnit does not get the chance to perform the cleanup operation. In Java, you can create a method named `finalize` for any class. This is a special method that the JVM will try to call when the object is garbage collected. It can be used by that object to clean up any resources it might be using. WebDriver is a resource. Maybe you can try to make your code more robust by overriding `finalize`?

FinalizeWebDriverIT.java

```
@Override  
protected void finalize() throws Throwable {  
    driver.quit();  
    super.finalize();  
}
```

Unfortunately, Java makes no promises about when, or even if, the `finalize` method will be called. This won't solve your problem, as you can still be left with an open browser.

A more robust option is to make sure the browser is closed by adding a JVM **shutdown hook** [1]. A shutdown hook is executed automatically when the JVM terminates.

ShutdownHookWebDriverIT.java

```
private WebDriver driver;

@Before
public void setUp() throws Exception {
    driver = new FirefoxDriver();
    Runtime.getRuntime().addShutdownHook(new Thread(driver::quit));
}
```

This is a very basic way to manage the lifecycle of the driver. It's not the best solution, for a few reasons:

- There will be a lot of duplicate code related to cleanup.
- A new browser is created for each test, but you might want to reuse the same browser to make your tests faster.

In the next section, you will see strategies to solve these problems.

Having a single place to supply drivers

What was your first thought about the duplicate code above? Extract it into a common abstract superclass? If you thought that, then you may have been about to couple your tests into a single unit dependent on that class. You were about to take a step toward making your test suite harder to maintain.

What you can do is have a class to encapsulate the creation and management of the driver. You can then use that class to inject your tests with a driver.

Injecting a driver

I've mentioned Dependency Injection in this chapter, but you've not yet seen any code. Injecting a driver into your test is the first step in separating the management of the driver from the test.

You want to reduce the amount of duplicate code for creating and quitting drivers.

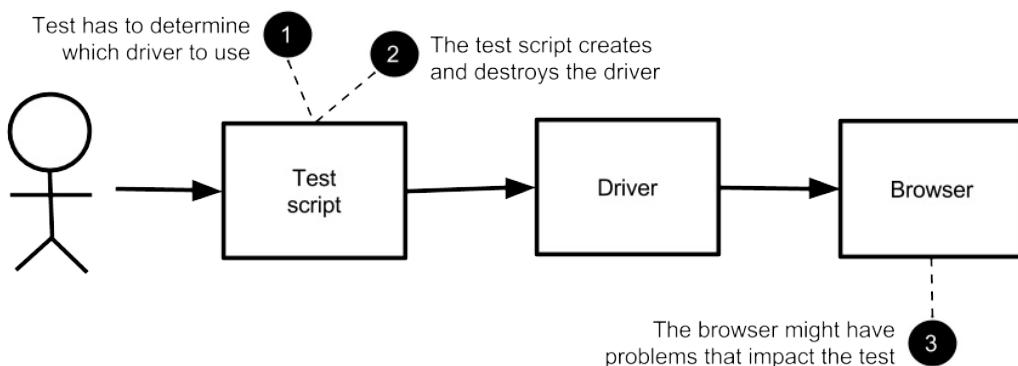
Use the Spring Framework [2] to create a Spring context, which is used to inject the driver into your tests.

Dependency Injection (DI) [3] is also known as **Inversion of Control (IoC)**. It is a design pattern that says that rather than an object finding its own dependencies, a third party determines them on its behalf.

Normally, if a class has a dependency on an instance of another class, it will create an instance of that class itself, using the `new` keyword. If you use DI, you state that you want the framework executing your code to choose an appropriate class for you.

In our case, this is a bit like saying "this test needs a web driver, but it is up to the test framework that runs the test to choose which web driver the test will run using." Not only does your test no longer need to create and configure a driver, but the responsibility for cleaning that driver up becomes a responsibility of the framework too. This leaves your test focused on testing.

Without Dependency Injection



With Dependency Injection

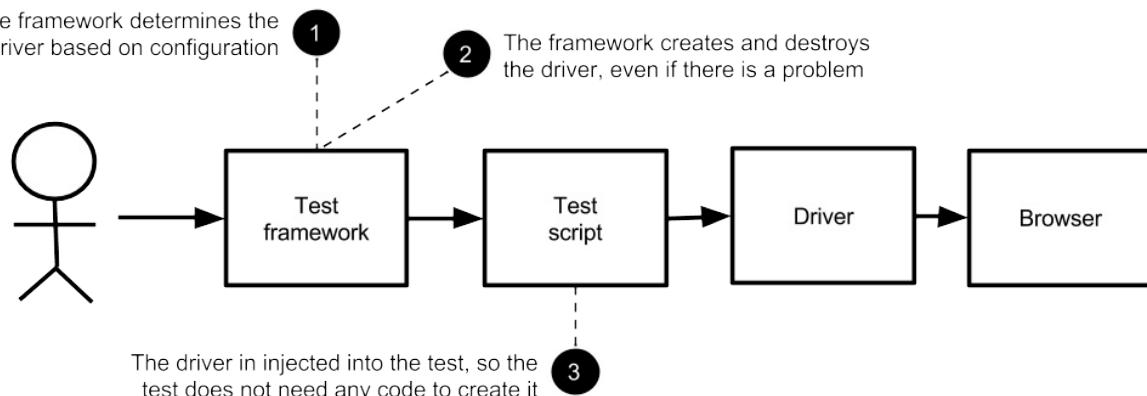


Figure 1. Dependency Injection

The `@Inject` annotation is a standard Java way for marking a field as one that needs injecting. By annotating a field on a test that is of class `WebDriver` with the `@Inject` annotation, you're saying to the software that runs the test, "please set this field to the driver you think it should be."

Using DI, you remove the need for tests to have any boilerplate code for getting dependencies, making them simpler, easier to maintain, and more flexible. Spring includes an excellent DI framework. You can use Spring to inject the tests. Spring provides a JUnit test runner that will run your tests, injecting dependencies into them.

The Spring Framework allows you to create a configuration for your tests. The configuration defines a **context**, which is essentially a set of Java objects (known as "beans") that can be reused for each test. It manages the beans so they are initialized when created, and destroyed when the context ceases to be used.

I'm going to assume you are using Maven to build your project. You will need a couple of additional dependencies for this technique:

[pom.xml](#)

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>4.2.5.RELEASE</version> (1)
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>4.2.5.RELEASE</version>
    <scope>test</scope>
</dependency>
```

1. You should update this version to the latest

Then, you can create a configuration file that defines the context for your tests:

[WebDriverConfig.java](#)

```

@Configuration
public class WebDriverConfig {

    @Bean
    public static PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer(); (1)
    }

    @Bean
    public DesiredCapabilities desiredCapabilities(
        @Value("${webdriver.capabilities.browserName:firefox}") String browserName (2)
    ) {
        return new DesiredCapabilities(browserName, "", Platform.ANY);
    }

    @Bean(destroyMethod = "quit") (3)
    public WebDriver webDriver(DesiredCapabilities desiredCapabilities) { (4)
        switch (desiredCapabilities.getBrowserName()) {
            case BrowserType.FIREFOX:
                return new FirefoxDriver(desiredCapabilities);
            case BrowserType.HTMLUNIT:
                return new HtmlUnitDriver(desiredCapabilities);
            default:
                throw new IllegalStateException("unknown browser " + desiredCapabilities.get
BrowserName());
        }
    }
}

```

1. This is a special bean that you need so that the `${...}` properties are supported.
2. Here you can use `@Value` to get a property for the browser. The `:firefox` part of the property indicates that the desired browser should default to Firefox.
3. If you need to clean up any beans once you've used the context, you need to tell Spring. Here it is told the `quit` method must be called for cleanup.
4. Return a desired `WebDriver` implementation based on the `getBrowserName()`

Spring XML Config vs. Spring Java Config

Spring provides two ways to define a context. The first way is using XML, but since Spring version 3, you can use Java Config. Java Config is much less verbose than XML, so we'll be using it in all the examples.

Finally, you can update your test to have the necessary annotations:

[InjectedDriverIT.java](#)

```

@RunWith(SpringJUnit4ClassRunner.class) (1)
@ContextConfiguration(classes = WebDriverConfig.class) (2)
public class InjectedDriverIT {
    @Inject
    private WebDriver driver; (3)

    @Test
    public void loadIndexPage() throws Exception {
        driver.get("http://localhost:8080/index.html");
    }
}

```

1. Tell JUnit to run the test with a custom runner: the Spring runner.
2. Tell Spring where the configuration is.
3. Indicate to Spring where to inject a driver.

Using Dependency Injection gives the control of which driver to use for the test to the configuration. If you want the configuration to return the Chrome driver, you can do this. The tests will run using that driver, but without any change to the tests themselves. WebDriver provides a class named `DesiredCapabilities` that indicates what kind of browser capabilities you want. One of those capabilities is `browserName`. You can change the browser you want to use for your tests using a system property. For example, to run the tests using the HTMLUnit driver:

```
mvn ... -Dwebdriver.capabilities.browserName=htmlunit
```

You will see that I have prefixed the property's name with the string `webdriver`. If you use a common prefix like this, then you are unlikely to find your property name is the same as that used by another library your application depends on.

Currently, this doesn't allow you to run remote web drivers, but you can expand your configuration by having a method to create remote drivers:

[framework/WebDriverConfig.java](#)

```

private WebDriver remoteDriver(URL remoteUrl, DesiredCapabilities desiredCapabilities) {
    return new Augmenter().augment(new RemoteWebDriver(remoteUrl, desiredCapabilities))
}; (1)
}

```

1. You need to augment your driver if you want to take screenshots.

The `Augmenter` class

The `RemoteWebDriver` class won't allow you to take screenshots as it does not implement the `TakesScreenshot` interface. You need to wrap it in an `Augmenter` if you want to take screenshots.

And another for local drivers:

[framework/WebDriverConfig.java](#)

```
private WebDriver localDriver(DesiredCapabilities desiredCapabilities) throws IOException {
    switch (desiredCapabilities.getBrowserName()) {
        case BrowserType.CHROME:
            return new ChromeDriver(desiredCapabilities);
        case BrowserType.FIREFOX:
            return new FirefoxDriver(desiredCapabilities);
        case BrowserType.HTMLUNIT:
            return new HtmlUnitDriver(desiredCapabilities);
        case BrowserType.SAFARI:
            return new SafariDriver(desiredCapabilities);
        default:
            throw new IllegalStateException("unknown browser " + desiredCapabilities.get
BrowserName());
    }
}
```

Here is the method that creates the driver for the tests to use:

[framework/WebDriverConfig.java](#)

```
@Bean(destroyMethod = "quit")
public WebDriver webDriver(
    @Value("${webdriver.remote:false}") boolean remoteDriver,
    @Value("${webdriver.remote.url:http://localhost:4444/wd/hub}") URL remoteUrl,
    DesiredCapabilities desiredCapabilities) throws Exception {

    return remoteDriver ?
        remoteDriver(remoteUrl, desiredCapabilities) :
        localDriver(desiredCapabilities);
}
```

Now you can change the driver to remote based on the property `webdriver.remote`, and the URL using `webdriver.remote.url`.

Tests running using Spring reuse the same beans. This means that the context can become "dirty." For example, cookies set as part of one test will remain for the next test. As cookies are often used for login, if the first test logged the user in but the second expected them to be logged out to start with, this will be a problem.

This can be partly addressed by using the special Spring annotation

`@Scope("prototype")`. This annotation tells Spring to create a new driver for every test. However, you might only need to use a driver whose cookies have been deleted, so you could add this method to your configuration:

framework/WebDriverConfig.java

```

@Bean
@Primary (1)
@Scope("prototype")
public WebDriver cleanWebDriver(WebDriver driver) throws Exception {

    driver.manage().deleteAllCookies(); (2)

    return driver;
}

```

1. Mark this bean as "primary." This means that it'll be used in preference to other beans.
2. Use the `deleteAllCookies` method to clean the driver.

This "clean" driver will be used for each test. But this might not be enough; the driver might be very dirty! For example, a pop-up might have been left open. You can resolve these more serious problems by annotating tests that make the driver very dirty with the `@DirtiesContext` annotation. This indicates to Spring that after these tests are run, the context should not be used again, and a fresh new one created and used for the next test. For example:

```

@Test
@DirtiesContext
public void dirtyTheDriver() throws Exception {
    driver.get("http://localhost:8080/popups.html");

    driver.findElement(By.linkText("Prompt")).click();
}

```

Making code run using base URLs

WebDriver requires absolute URLs. An absolute URL is one that includes the protocol, the host, and optionally the port; for example, <http://localhost:8080/my-page.html>. But what if you want to run the tests locally against one server, then on your CI server, making requests to a different server? You might want to specify a single **base URL** in one place, and then use that URL in all of your tests.

Injecting a base URL

The following technique makes use of Dependency Injection, but this time you inject a URL rather than the driver.

You want to run the same code against both a local and a remote web server.

Inject a base URL into your tests.

Define the base URL in the Spring configuration file. Add the following lines:

[injectingbaseurl/WebDriverConfig.java](#)

```
@Bean  
public URI baseUrl(@Value("${webdriver.baseUrl:http://localhost:8080}") URI value) {  
    return value;  
}
```

You can inject this into your code:

[InjectedBaseUrlIT.java](#)

```
@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(classes = WebDriverConfig.class)  
public class InjectedBaseUrlIT {  
    @Inject  
    private WebDriver driver;  
  
    @Inject  
    private URI baseUrl; (1)  
  
    @Test  
    public void loadIndexPage() throws Exception {  
        driver.get(baseUrl + "/index.html");  
    }  
}
```

1. URL is injected here.

You can run this test with alternative URLs from your terminal by using Maven and changing the property:

```
mvn failsafe:integration-test \
-Dwebdriver.baseUrl=http://mytestserver
-Dit.test=InjectedBaseUrlIT
```

I hope you can see how using Dependency Injection means that you do not need to change your tests when the setup or configuration needs to change. The website is hosted somewhere else? You want to change the browser? No problem! All you need to do is update one property, and off you go!

Taking a screenshot when a test finishes

It can be really helpful to take screenshots as part of a normal test cycle, for several reasons:

1. A test fails and you cannot determine the reason it failed because the browser closed before there was a chance to see what happened.
2. A test fails on your CI server, but passes locally and you want more information to diagnose it.
3. A test does not provide enough information to diagnose it.

Using the Spring test listener to take a screenshot when a test finishes

WebDriver provides a way to take and save a screenshot. Spring has a mechanism that allows us to have code run each time a test finishes. This technique combines those two features to take a screenshot whenever a test fails.

You want to take a screenshot of the browser at the end of every test automatically.

To solve this problem, you'll need to:

1. Expose the web driver's ability to take screenshots.
2. Take a screenshot when a test finishes.

3. Copy that screenshot somewhere safe, so it can be accessed later on.

ScreenshotTaker.java

```
public class ScreenshotTaker extends AbstractTestExecutionListener { (1)
    @Override
    public void afterTestMethod(TestContext testContext) throws Exception {
        TakesScreenshot takesScreenshot = (TakesScreenshot)testContext.getApplicationConte
xt()
            .getBean(WebDriver.class); (2)
        File screenshot = takesScreenshot.getScreenshotAs(OutputType.FILE); (3)
        File file = new File("target",
            testContext.getTestClass().getName() + "_" + testContext.getTestMethod().ge
tName() + ".png"); (4)
        FileUtils.deleteQuietly(file);
        FileUtils.moveFile(screenshot, file); (5)
        System.out.println("saved screenshot as " + file);
    }
}
```

1. Extend the `AbstractTestExecutionListener` class.
2. Get the web driver from the Spring context and cast it to `TakesScreenshot`.
3. Save the screenshot as a file.
4. Choose a suitable name for the screenshot based on the test class and name.
5. Move the screenshot to somewhere safe.

Finally, you need to annotate your test with the `@TestExecutionListeners` annotation, so that Spring is aware of the listeners.

ScreenshotIT.java

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = WebDriverConfig.class)
@TestExecutionListeners(listeners = {ScreenshotTaker.class, DependencyInjectionTestExecutionL
istener.class}) (1)
public class ScreenshotIT {
    // ...
}
```

1. Add the `ScreenshotTaker` and `DependencyInjectionTestExecutionListener` to the `@TestExecutionListeners` annotation.

You need to add both `TestExecutionListeners` and `DependencyInjectionTestExecutionListener` to the `@TestExecutionListeners` annotation. The first listener takes the screenshot and the second one performs the dependency injection. `DependencyInjectionTestExecutionListener` is normally added by default, but if you need to add more, then you need to override it.

Taking screenshots is a great aid to debugging. You will want to tie in with the testing process to take them, and you want to make sure you name the files in a helpful fashion. The above example uses the test name to generate the name for the screenshot. If your test is running on a CI server, you will need to ensure it makes them available to view and does not delete them as part of its cleanup process.

I am willing to bet that, like me, you have had instances where a test has failed on the CI server, but not enough information was output to help diagnose the problem. You then ran the test locally only to find it would always pass. Worse still, maybe you also have had to diagnose a test that failed on the CI only sporadically, but always passed locally! Having a screenshot taken at the time of the failure can make it much easier to understand what the cause was.

Good diagnostics are key to any good testing framework. Here is a great example from WebDriver itself:

```
org.openqa.selenium.NoSuchElementException: Unable to locate element using css
For documentation on this error, please visit: http://seleniumhq.org/exceptions/no_suc
h_element.html
Build info: version: '2.45.0', revision: '5017cb8e7ca8e37638dc3091b2440b90a1d8686f', tim
e: '2015-02-27 09:10:26'
System info: host: 'alex-collinss-macbook.local', ip: '192.168.59.3', os.name: 'Mac OS X', os
.arch: 'x86_64', os.version: '10.7.5', java.version: '1.8.0_20'
Driver info: driver.version: HtmlUnitDriver
at org.openqa.selenium.htmlunit.HtmlUnitDriver.findElementByCssSelector(Ht
mlUnitDriver.java:995)
at org.openqa.selenium.By$ByCssSelector.findElement(By.java:425)
```

From this, I have information about:

1. The running code
2. The computer the test is running on (OS and so on)
3. Where to look for more information about the `NoSuchElementException` error

Taking screenshots as part of the test lifecycle in a convention-based manner, rather than manually or ad hoc, means that everyone on your team will know where the screenshots can be found, and therefore will be able to diagnose failing tests quicker. This will reduce the maintenance cost of your test suite.

Summary

- WebDriver doesn't provide any way to manage the driver; you need to do this yourself.
- Browsers can remain open if you do not quit them. You need to take special action to close them. Otherwise, in the worst case, you might even crash a computer by opening too many browsers.
- Dependency Injection allows you to have your driver managed centrally. Spring can manage and clean the driver up once it becomes very dirty.
- Dependency Injection can also provide you with a way to inject a base URL.
- Screenshots can be taken while running the tests and they are useful as a debugging aid when test fails

This is the end of part one. We've covered all the fundamentals of WebDriver: locating, interacting, and checking elements; page objects; what to do when things go wrong; and managing WebDriver. In the next part we'll dig deeper into the APIs.

1.

<http://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#addShutdownHook-java.lang.Thread>

2. <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html>

3. https://en.wikipedia.org/wiki/Dependency_injection

Chapter 8: Windows, pop-ups, and frames

This chapter covers

- Encapsulating new windows
- Interacting with pop-ups
- Accessing inline frames

Part 1 of this book will have given you many strategies to deal with most of the common automation techniques that you need for working with web applications.

In this chapter we will go over working with windows, pop-ups, and inline frames. We will cover techniques for accessing windows and frames reliably, and dealing with pop-up alerts. Imagine you have a test that opens an alert, but then fails due to the contents of the alert, and does not close the alert afterward. This will result in an alert being left open that might interfere with another piece of code. We'll cover a technique to deal with this situation.

By the end of this chapter you will have learned how to encapsulate both window and pop-up handling into utility classes, as well as ways to interact with frames.

Finding the window a page opens

When automating a page, you might find that the browser opens a new window. This might be a confirmation dialog box, an advertisement, or simply an external link.

New windows can be opened either if the HTML uses the `target` attribute, as in

```
<a href="new-window.html" target="new-window-name">  
  Open A New Window</a>
```

or using JavaScript's `window.open()` function, as in

```
<a href="#" onclick="window.open('new-window.html');">  
  Open A New Window</a>
```

When you are working with new windows you are likely to find yourself doing the steps in figure [Windows - localhost:8080/open-a-new-window.html](#).

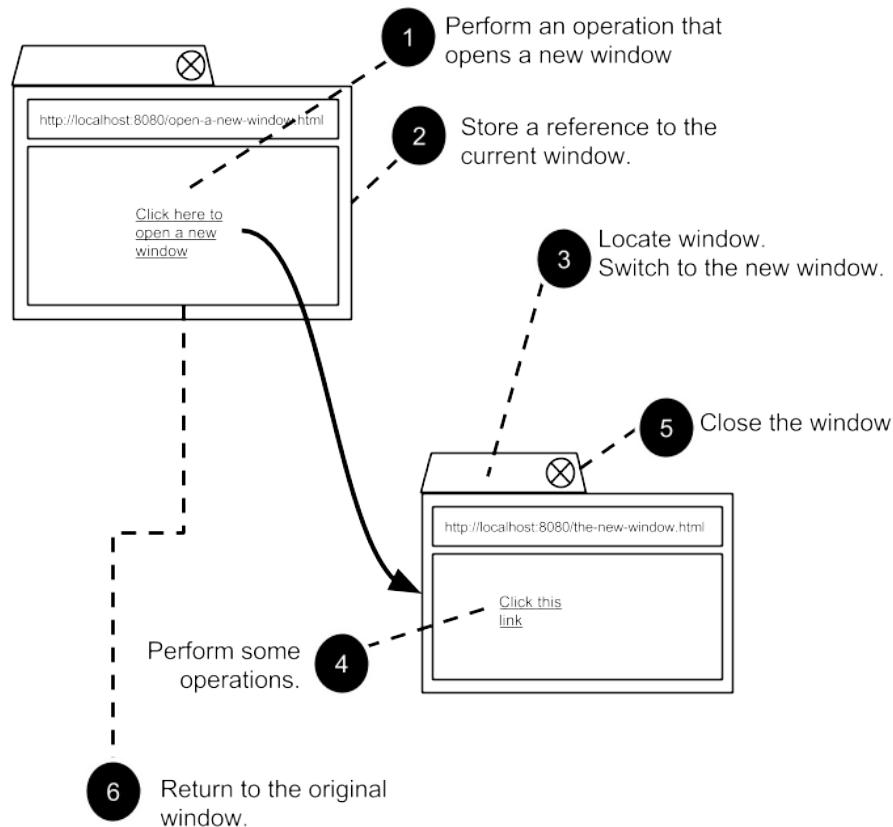


Figure 1. Windows - localhost:8080/open-a-new-window.html

The steps are

1. Execute an operation that you expect to result in a new window.
2. Store a reference to the current window.
3. Locate the new window.
4. Switch to the new window.
5. Perform operations on the new window.
6. Execute an operation that closes the new window.
7. Switch back to the original window.

Dirty windows

When working with the new window, you may find that your script throws an exception and leaves the browser in a "dirty" state, with multiple windows open that slow down the machine, or result in unpredictable behavior in code that runs later. We want to make sure that, even in the event of failure, we clean up after ourselves by closing any opened windows.

You can examine the open windows using `driver.getWindowHandles()`. This returns a list of **window handles**. Window handles are hexadecimal strings, [1] for example

`fef0ba8a-6400-304a-b46b-b8919ba1d354`, and do not provide enough information on their own to differentiate which window is which. There are some good strategies to find the right window:

- If you think there are only two open windows, then only one window can be the newly opened window, and it's not the one that you are currently on.
- Locate the window based on its name.
- Switch to another window and use information that web driver can tell you about that window (for example, it's title) to determine if it is the window you want.

What would these steps look like as code?

NewWindowIT.java

```
driver.get("http://localhost:8080/open-a-new-window.html");

String originalWindowHandle = driver.getWindowHandle(); (1)
try { (2)

    driver.findElement(By.tagName("a")).click(); (3)

    for (String windowHandle : driver getWindowHandles()) { (4)
        if (!windowHandle.equals(originalWindowHandle)) { (5)
            driver.switchTo().window(windowHandle); (6)
            break;
        }
    }
    try {
        (7)
    } finally {
        driver.close(); (8)
    }
} finally {
    driver.switchTo().window(originalWindowHandle); (9)
}
```

1. Store the original window handle.
2. Wrap the operations on the new window in a try/finally block.
3. Execute an operation to open a new window.
4. Iterate over all open windows.
5. Find a window that is not the current one.
6. Switch to the new window.
7. Perform operations on the new window.
8. Make sure the new window is closed.
9. Switch back to the original window.

If you look at the HTML source code from earlier, you can see that the link has a `target` attribute:

```
<a href="new-window.html" target="new-window-name"
    >Open A New Window</a>
```

The `target` attribute indicates the name of the window it will open. You can use the code in the following listing to find that window.

NewWindowIT.java

```
driver.get("http://localhost:8080/open-a-new-window.html");

try {
    driver.findElement(By.tagName("a")).click();

    driver.switchTo().window("new-window-name");

    try {
        assertEquals("You Are In The New Window", driver.findElement(By.tagName("h1"))
            .getText());
    } finally {
        driver.close();
    }
} finally {
    driver.switchTo().defaultContent();
}
```

If the "finding the window that is not the original window" strategy does not work (e.g. in the rare case there are multiple open windows), you can use a feature of the window to identify it. This code sample looks for the first window that has an `h1` heading of "You Are In The New Window".

NewWindowIT.java

```
driver.get("http://localhost:8080/open-a-new-window.html");

String originalWindowHandle = driver.getWindowHandle();

driver.findElement(By.tagName("a")).click();

try {
    for (String windowHandle : driver.getWindowHandles()) {
        driver.switchTo().window(windowHandle);
        if (driver.findElement(By.tagName("h1")).getText()
            .equals("You Are In The New Window")) { (1)
            break;
        }
    }
}

assertEquals("You Are In The New Window", driver.findElement(By.tagName("h1")).get
Text());

driver.close();
} finally {
    driver.switchTo().window(originalWindowHandle);
}
```

1. Find a window that has the text you want.

As the filter switches to the window as part of the operation, you do not need to switch to it afterward.

Both these approaches have a fair amount of boilerplate code. You would probably just like to focus on the important things:

- Opening the window
- Identifying the window to switch to
- Performing operations on the new window

You've seen the basics of opening windows and tidying up after yourself. Let's now look at encapsulating that into a useful utility class.

Encapsulating window handling

With this technique you will see how you can encapsulate the three operations into a single utility class.

Code that has to deal with new windows is verbose and error-prone.

Encapsulate the handling of new windows to represent these three actions:

- Opening the window:

```
driver.findElement(By.tagName("a")).click()
```

- Identifying the window:

```
!windowHandle.equals(originalWindowHandle)
```

- Operating on the window:

```
assertEquals("Thank You!", driver.findElement(By.tagName("h1")).getText());
```

You can put this together into a class. We'll create a class named `WindowHandler`. To use this, you override the `openWindow` and `useWindow` methods.

[WindowHandlerIT.java](#)

```
new WindowHandler(driver) {
    @Override
    public void openWindow(WebDriver driver) { (1)
        driver.findElement(By.tagName("a")).click();
    }

    @Override
    public void useWindow(WebDriver driver) { (2)
        assertEquals("You Are In The New Window", driver.findElement(By.tagName("h1"))
            .getText());
    }
}.run(); (3)
```

1. This is a method that will open the window.
2. This is a method that will use the window.
3. This will open the window, use it, and then clean up afterward.

Let's put the underlying `WindowHandler` class together.

WindowHandler.java

```
public abstract class WindowHandler {  
    private final WebDriver driver;  
  
    public WindowHandler(WebDriver driver) {  
        this.driver = driver;  
    }  
  
    protected abstract void openWindow(WebDriver driver); (1)  
  
    protected boolean isExpectedWindow(WebDriver driver, String originalWindowHandle) {  
        return !driver.getWindowHandle().equals(originalWindowHandle); (2)  
    }  
  
    public abstract void useWindow(WebDriver driver); (3)  
  
    public void run() {  
        String originalWindowHandle = driver.getWindowHandle();  
  
        openWindow(driver); (4)  
        try {  
            for (String windowHandle : driver.getWindowHandles()) {  
  
                driver.switchTo().window(windowHandle);  
  
                if (isExpectedWindow(driver, originalWindowHandle)) {  
  
                    useWindow(driver);  
  
                    if (!driver.getWindowHandle().equals(originalWindowHandle)) { (5)  
                        driver.close();  
                    }  
  
                    return;  
                }  
            }  
            throw new IllegalStateException("unable to find correct window");  
        } finally {  
            driver.switchTo().window(originalWindowHandle);  
        }  
    }  
}
```

1. Override this method to perform an action to open the window.

2. You can use the default method, or just find a window that was not the opening window.
3. Override this method to use the window (for example, clicking on it or asserting on it).
4. This code is largely the same as the code we used previously.
5. Only close the window if using it didn't result in it being closed already.

This class looks long, but it can be reused whenever you want to deal with opened windows. We wrap up the opening of the window, identifying the opened window, and then operating on the window. By putting the executed code into a method, we can then perform the necessary cleanup afterward.

Modal pop-ups

There are a couple of types of modal pop-up: **JavaScript**, **HTTP authentication**, **synthetic**, and **system**. They are modal because they are designed to prevent the user from performing any operation until they have dismissed the dialog box. They are typically used to make sure that the user has performed some action before they continue; for example, requiring confirmation to make sure they realize that what they are about to do cannot be undone.

JavaScript pop-ups

There are three types of JavaScript pop-up you may be familiar with: **alert**, **confirm**, and **prompt**.

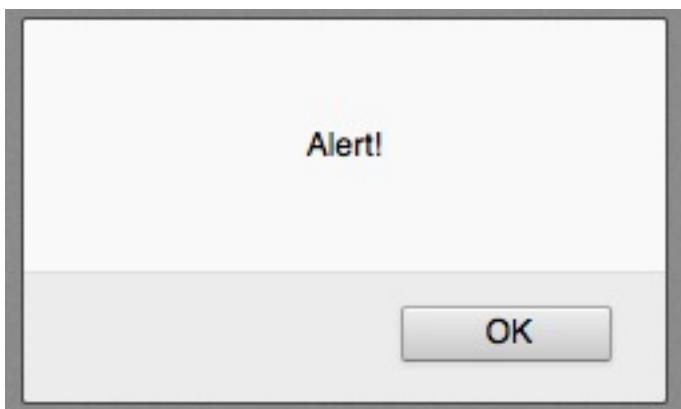


Figure 2. JavaScript Alert - <http://localhost:8080/popups.html>

The HTML source code would show you the use of the JavaScript `alert` function

```
<a href="#" onclick="alert('Alert!');">Alert</a>
```

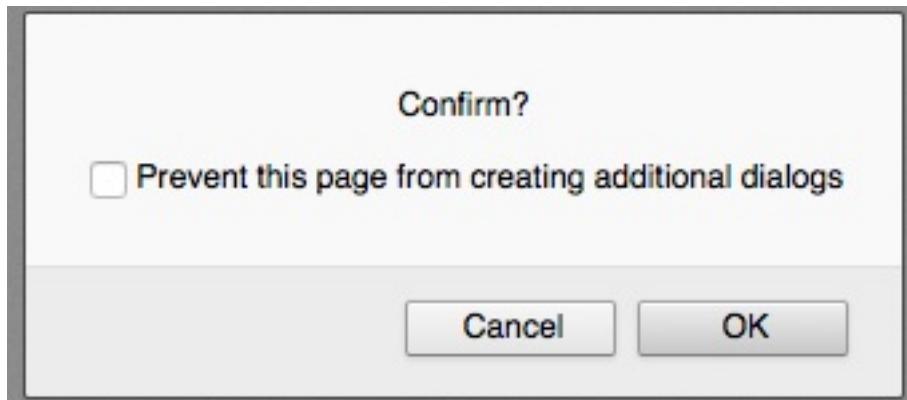


Figure 3. JavaScript Confirm - <http://localhost:8080/popups.html>

For a confirmation popup, you will often be able to see the use of the JavaScript `confirm` function:

```
<a href="#" onclick="confirm('Confirm?');">Confirm</a>
```

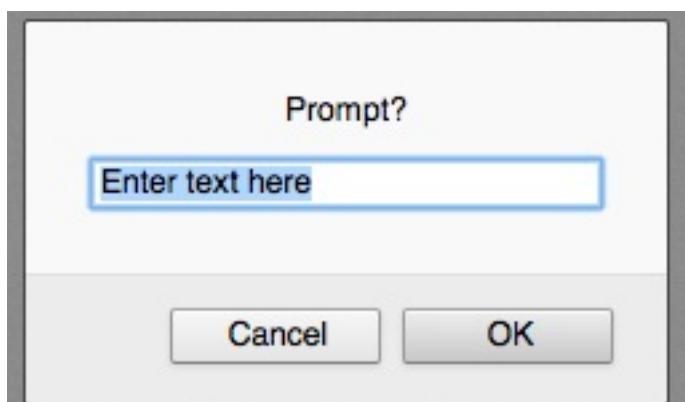


Figure 4. JavaScript Prompt - <http://localhost:8080/popups.html>

```
<a href="#" onclick="prompt('Prompt?', 'Enter text here');">Prompt</a>
```

To dismiss an alert, confirm, or prompt pop-up (that is, click "OK"):

```
driver.switchTo().alert().dismiss();
```

If you want to accept a confirm box or prompt:

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

If you want to enter text into (or get text from) a prompt:

```
driver.switchTo().alert().sendKeys("text")
driver.switchTo().alert().getText();
```

The Safari driver does not support alerts. They are automatically dismissed.

You may find that you get a `NoAlertPresentException` error. To wait for the alert, you can use an instance of the `WebDriverWait` class with the `ExpectedConditions.alertIsPresent` condition.

JavaScriptAlertIT.java

```
new WebDriverWait(driver, 5).until(ExpectedConditions.alertIsPresent());
```

HTTP authentication pop-ups

Perhaps less common these days is the HTTP authentication [2] pop-up. You can only dismiss this using a username and password. Unfortunately, WebDriver browser implementations do not currently support this.

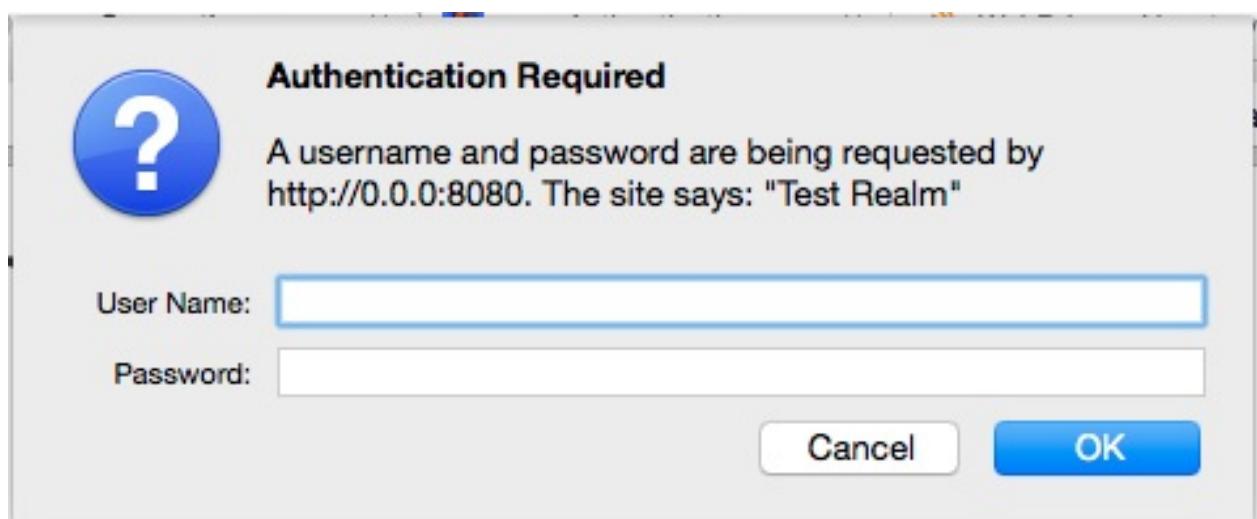


Figure 5. HTTP Authentication Pop-up

The authentication is enforced by the addition of HTTP headers. HTTP headers cannot be seen in the normal HTML source views provided by most browsers. You can find them in the network view in Firefox, under Response Headers.

You can see in figure [Firefox Network Panel](#) that the HTTP response code was "401 Unauthorized," which indicates that authentication is required. Also, there is a "WWW-Authenticate" header, which tells the browser to show the HTTP authentication pop-up.

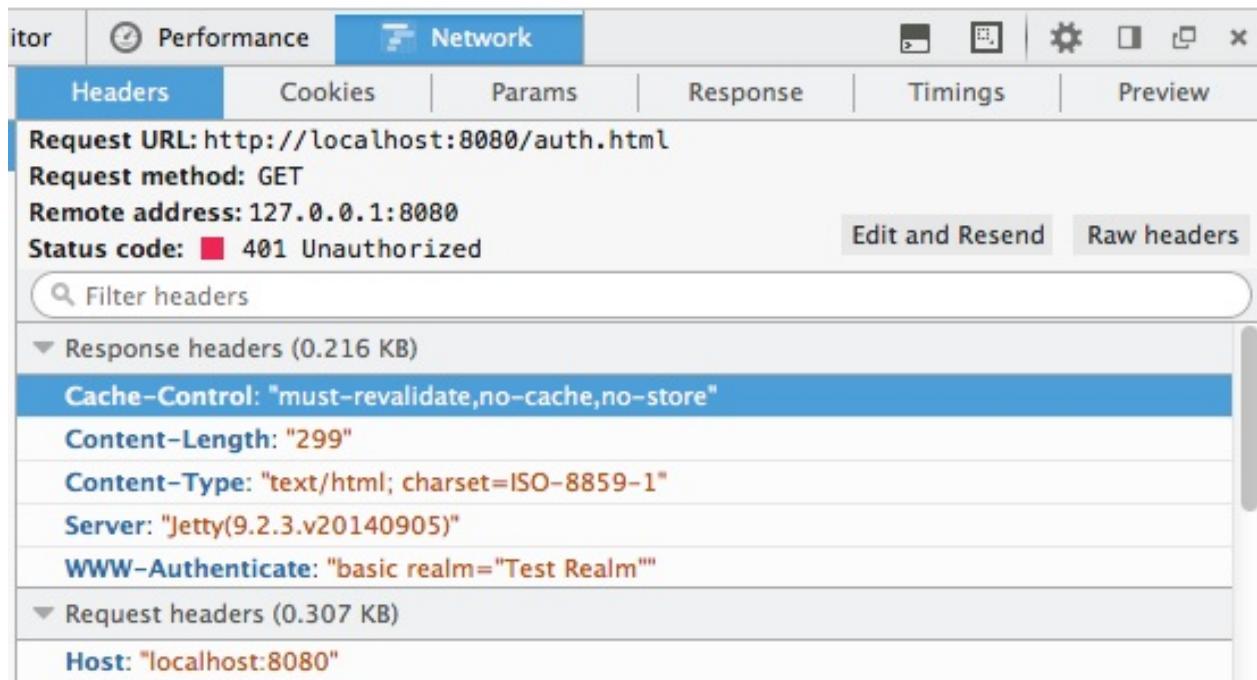


Figure 6. Firefox Network Panel

Unfortunately, as this is a system pop-up, WebDriver cannot enter anything into it. To deal with this pop-up, you need to pass the username and password in the URL. You do this by inserting the username and password before the hostname. If the username is "admin" and the password is "secret", then you need to add "admin:secret@" to the URL, as in the following listing.

HttpAuthenticationIT.java

```
driver.get("http://admin:secret@localhost:8080/auth.html");

assertEquals("You Are Logged In", driver.findElement(By.tagName("h1")).getText());
```

This way of setting the password reveals it to anyone using your computer. Make sure you don't accidentally reveal your bank account password by doing this!

Synthetic pop-ups

Bootstrap and other frameworks provide JavaScript pop-ups that do not use the native support. These pop-ups are actually HTML overlaid onto the page.



Figure 7. Synthetic Modal Pop-up - <http://localhost:8080/popups.html>

We can inspect the source code for this to see the relevant HTML.

```
<div class="modal fade" id="basicModal" tabindex="-1" role="dialog"
     aria-labelledby="basicModal" aria-hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal"
               aria-hidden="true">&times;</button>
        <h4 class="modal-title" id="myModalLabel">Modal</h4>
      </div>
      <div class="modal-body">
        <p><input class="span12" type="text"></p>
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-primary">OK</button>
        <button type="button" class="btn btn-default"
               data-dismiss="modal">Cancel</button>
      </div>
    </div>
  </div>
</div>
```

Looking at this, you can see that the modal is actually composed of many different HTML elements, such as divs and buttons.

Creating a Page Object for synthetic modals

As a synthetic modal contains many different elements, but encapsulated on behavior, it is a good candidate for a Page Object. Perhaps you do not want to know what kind of modal you are using, but you want to hide the nitty-gritty of how to interact with it

— after all, you just want to inspect it, then dismiss it! The next technique looks at how to encapsulate a synthetic modal into a Page Object.

You want a unified way to identify modal pop-ups, inspect them, and dismiss them.

Create a Page Object that encapsulates the pop-up by implementing the `Alert` interface and provides an `ExpectedCondition` to identify it.

Let's look at how you can implement the condition first.

Modals.java

```
public static ExpectedCondition<Alert> modalsDisplayed() { (1)
    return new ExpectedCondition<Alert>() {

        @Override
        public Alert apply(WebDriver driver) {
            List<WebElement> bootstrapModal = driver.findElements(By.className("modal-dialog"));
            List<WebElement> otherModal = driver.findElements(By.className("other-modal"));

            return !bootstrapModal.isEmpty() (2)
                ? new BootstrapModal(bootstrapModal.get(0))
                : !otherModal.isEmpty()
                    ? new OtherModal(otherModal.get(0))
                    : null;
        }
    };
}
```

1. A static method that returns an `ExpectedCondition`.
2. Look for a Bootstrap modal first, then look for others.

The modal might not be visible to the user, so you should check for both presence and visibility.

This solution provides a single class for locating modals. The modal you have might be written using the Bootstrap framework, but there are other frameworks, and you could extend this to support those frameworks by searching for their types as well.

You can implement each type of synthetic alert individually.

BootstrapModal.java

```
public class BootstrapModal implements Alert { (1)
    private static final By INPUT_SELECTOR
        = By.cssSelector("input[type='text']");
    private final SearchContext searchContext;

    public BootstrapModal(SearchContext searchContext) { (2)
        this.searchContext = searchContext;
    }

    @Override
    public void dismiss() {
        searchContext.findElement(By.cssSelector("button.btn-default"))
            .click(); (3)
    }

    @Override
    public void accept() {
        searchContext.findElement(By.cssSelector("button.btn-primary"))
            .click(); (4)
    }

    @Override
    public String getText() {
        return searchContext.findElement(INPUT_SELECTOR)
            .getAttribute("value"); (5)
    }

    @Override
    public void sendKeys(String keysToSend) {
        searchContext.findElement(INPUT_SELECTOR).sendKeys(keysToSend);
    }

    @Override
    public void authenticateUsing(Credentials credentials) {
        throw new UnsupportedOperationException();
    }
}
```

1. Implementing the `Alert` interface will ensure other developers will understand how to use it.
2. Accept just the part of the page we are interested in.
3. The cancel button.
4. The OK button.
5. The text input.

Implementing a Page Object for a synthetic modal is reasonably straightforward. Using the same interface, `Alert`, as other browser alerts means that you can abstract away the details of the individual alert from the code using it, arguably making for easier-to-read code. This also can mean that, if the modal framework used changes, the automation code does not have to.

A couple of words of warning. While both native and synthetic dialog will prevent a user accessing the page, a synthetic may not block WebDriver from accessing the page. It will be possible for your code to find parts of the page a user would not be able to, but not interact with them.

```
@Test
public void demonstrateAccessToUnclickableElement() throws Exception {

    driver.get("/popups.html");

    driver.findElement(By.linkText("Modal")).click();

    new WebDriverWait(driver, 2).until(Modals.modalIsDisplayed());

    WebElement openModalButton = driver.findElement(By.linkText("Modal")); (1)

    try {
        openModalButton.click(); (2)
        fail("should not be able to click when modal is displayed");
    } catch(WebDriverException e) {
        assertThat(e.getMessage(),
                   containsString("Other eleme
nt would receive the click"));
    }
}
```

1. We can still locate the element, even thought the modal dialog is visible.
2. But, when we try to click, we'll get an exception.

There is get a message to indicate that the element is not-clickable. If this was an alert, then there would get a clear message saying an alert is open.

This can make for hard-to-identify bugs if care is not taken when working with synthetic modals. For example, if a native alert pop-up appears when your test does not expect it, you will get errors when you try to interact with other elements. If this happens with a synthetic alert, then your test may pass when it shouldn't have.

System pop-ups

You may find that the browser sometimes opens up a system pop-up. WebDriver does not provide a way to interact with these, and you want to make sure all your techniques work on both locally running browsers (where you could use Java's built-in `Robot` class) and remotely running browsers (where you cannot).

The most effective technique to deal with them is to make sure that the browser is configured to prevent them from occurring in the first place. This can often be achieved from the browser's settings.

To find out more about configuring browsers, please read chapter 11, which cover many aspects of drivers.

Inline frames

As single-page web apps have become more popular, embedded **inline frames** (a.k.a. "iframes," or sometimes just "frames") are another feature of web pages that are seen less often in the wild. Yet I'm sure you will encounter them sooner or later, for example, in applications involving the following:

- Advertisements
- WYSIWYG editors
- OAuth authentication
- 3D Secure (3DS) banking

Inline frames are very similar to new windows. They contain fully formed pages, they can be from another website, and you can have more than one open. This means some of the techniques that apply to windows also apply to frames.

Inline Frames

Frame 1
Content

Frame 2
Content

Figure 8. Inlines Frames

The HTML code looks like this:

```
<iframe src="frame-1-content.html" name="frame-1"></iframe>
<iframe src="frame-2-content.html" name="frame-2"></iframe>
```

You may find that the iframes are named, which makes switching to them straightforward:

```
driver.switchTo().frame("frame-1");
```

Once you are done, you can switch back to the original page with `defaultContent()`:

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

You might wish to use a `try/finally` block, in case any of the operations you perform fail and you are left focused on a page you do not want to be focused on:

```
try {
    driver.switchTo().frame("frame-1");
} finally {
    driver.switchTo().defaultContent();
}
```

Summary

- Newly opened windows can be accessed using `driver.switchTo().window(..)`. You can return to the original window using `driver.switchTo().defaultContent()`.
- If a window or frame is opened, your code should switch back to the original page when done interacting with that window.
- There are a variety of different pop-ups that you can interact with. Some are native to the browser, but you can also have synthetic pop-ups written using a JavaScript framework.
- Interacting with pop-ups can be encapsulated into a utility class.
- Inline frames can be accessed using the `driver.switchTo().frame(..)` method.

In the next chapter we will look at a number of less common web page features, and how to test them.

1. <https://en.wikipedia.org/wiki/Hexadecimal>
2. http://en.wikipedia.org/wiki/Basic_access_authentication

Chapter 9: Unicorns and other beasts: Exotic features of web pages

This chapter covers

- Using action chains
- Advanced form interaction
- Working with different types of tooltip

The first part of this book has enough techniques to test many web pages. But there are a few cases that we have not covered yet – the unicorns of web application testing.

In this chapter we'll look at using action chains to achieve such goals as copy and paste, or moving and dragging with the mouse. We'll cover advanced form interaction, such as multiple selects, and how to upload files to a server. We'll see how WebDriver exposes browser cookies, which can be very handy when trying to debug certain issues. Finally, there are several types of tooltip, and we'll look at interacting with them.

At the end of this chapter you will have learned how to use action chains to perform complex actions, how to upload files, and how to examine cookies and tooltips.

Advanced user interaction with action chains

An **action chain** is a way to chain several user interactions together. It's useful when you have to perform complex actions that the `WebElement` class does not directly support, such as:

- Typing keyboard "chords": holding down a "modifier key" such as Ctrl, then typing some text, and finally releasing the modifier key
- Copying and pasting: for example, selecting the text of an input, pressing Ctrl-C, moving to another input or element, and pressing Ctrl-V
- Hovering over an element to see if a tooltip appears or the element otherwise changes as expected
- Using the mouse to drag and drop

- Dragging elements by N pixels, for example, to drag a control or resize a text area
- Right-clicking using the mouse

The primary class for building and then performing a chain of actions is `Actions`. To use it, create a new instance from your driver, add some actions, and finally invoke the `perform` method. For example:

ExampleIT.java

```
driver.get("/login.html");

new Actions(driver)
    .sendKeys(driver.findElement(By.name("email")), "john.doe@email.com")
    .sendKeys(driver.findElement(By.name("password")), "secret")
    .click(driver.findElement(By.cssSelector("input[type='submit']")))
    .perform();
```

Most methods on the `Actions` class have two versions, one that takes a `WebElement` as the first parameter, and another that requires you to focus on the element beforehand. The preceding test can also be written as follows:

ExampleIT.java

```
new Actions(driver)
    .moveToElement(driver.findElement(By.name("email")))
    .click()
    .sendKeys("john.doe@email.com")
    .moveToElement(driver.findElement(By.name("password")))
    .click()
    .sendKeys("secret")
    .moveToElement(driver.findElement(By.cssSelector("input[type='submit']")))
    .click()
    .perform();
```

Tip

You might find it easier to read action chains if each action is on a single line.

There are a number of keyboard and mouse actions available to you (see table 9.1).

Table 1. Actions

Action	Description
<code>keyDown(Keys theKey)</code>	Press down a modifier key (such as Ctrl, Alt, or Shift).
<code>keyUp(Keys theKey)</code>	Release a modifier key.
<code>sendKeys(CharSequence... keysToSend)</code>	Send a series of keys.
<code>release()</code>	Release the left mouse button.
<code>click()</code>	Click on an element.
<code>doubleClick()</code>	Double-click on an element.
<code>contextClick()</code>	Right-click on an element, revealing any context menu.
<code>clickAndHold()</code>	Click the mouse button on an element, and hold it down.
<code>moveToElement(WebElement toElement)</code>	Move the mouse to the center of an element.
<code>moveToElement(WebElement toElement, int xOffset, int yOffset)</code>	Move the mouse to an element, but with an offset from the top-lefthand corner.
<code>moveByOffset(int xOffset, int yOffset)</code>	Move the mouse.
<code>dragAndDrop(WebElement source, WebElement target)</code>	Drag one element to another.
<code>dragAndDropBy(WebElement source, int xOffset, int yOffset)</code>	Drag an element by a specific offset.

Mobile automation

If you are doing mobile testing, then there is another class named `TouchActions` that provides similar methods for touchscreen devices – methods for actions such as touch, tap, double-tap, long press, or flick.

OK. So that was an overview of the APIs, but what are they useful for? One use is copying from one input to another. To do this, you need to use the correct modifier key with the copy key and the paste key. On Windows and Linux the modifier key is the Ctrl key, but on Mac OS X it's the Cmd key. This means there is a little extra logic necessary to determine which operating system you are using and therefore which modifier key to use.

[CopyAndPasteIT.java](#)

```
driver.get("/registration-form.html");

JavascriptExecutor javascriptExecutor = (JavascriptExecutor) this.driver;
String userAgent = (String) javascriptExecutor.executeScript("return navigator.userAgent");
(1)
Keys modifier = userAgent.contains("Mac OS X") ? Keys.COMMAND : Keys.CONTROL; (2)

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

new Actions(driver)
    .sendKeys(email, "john.doe@email.com")
    .sendKeys(email, Keys.chord(modifier, "a", "c")); (3)
    .sendKeys(password, Keys.chord(modifier, "v"))
    .perform();
```

1. Use `JavascriptExecutor` to get the browser's operating system
2. If you are using OS X then you need to use the Command key instead of the Control key
3. Use `Keys.chord(..)` to do a Ctrl-A followed by a Ctrl-C (or or Cmd-A and Cmd-C on OS-X)

You can use action chains to move the mouse over an element:

[MouseHoverIT.java](#)

```
driver.get("/mouse-hover.html");

new Actions(driver)
    .moveToElement(driver.findElement(By.id("target")))
    .perform();
```

Drag-and-drop is where you can move an element from one place to another using the mouse. The `Actions` class provides a method for that:

DragAndDropIT.java

```
driver.get("/drag-and-drop.html");

new Actions(driver)
    .dragAndDrop(
        driver.findElement(By.id("move")),
        driver.findElement(By.id("drop")))
    .perform();
```

Now you've seen a number of use cases for action chains. When you use them a lot, you may find out that you sometimes get a `StaleElementReferenceException`. This will be because you normally have to locate elements before you create the action chain. As you execute the chain, the page changes, and elements can become stale.

Lazy element for action chains

Action chains require you to locate elements prior to performing the sequence of actions. On certain pages, the element might become stale before part of the action is performed. In this case you want to locate the element lazily. This can be done by creating a lazy `WebElement` class.

You want to use action chains on elements that cannot be located before part of the chain is executed.

Use a **lazy element** that locates the element each time a method is called. This is similar to the mechanism that the `PageFactory` class uses.

To demonstrate this problem, we've created a page in the test app. The page at <http://localhost:8080/stale-elements.html> has a button that, when clicked, is removed from the page, and an identical button is added in its place. A second click results in a `StaleElementReferenceException`. Let's have a look at an example of a test that uses this problematic button.

LazyActionChainIT.java

```

@Test(expected = StaleElementReferenceException.class)
public void staleElementProblem() throws Exception {

    driver.get("/stale-elements.html");

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

    new Actions(driver)
        .click(button)
        .click(button) (1)
        .perform();
}

```

1. The second action will result in a `StaleElement` exception

You can create a class that implements the `WebElement` interface, but actually locates the element each time a method is invoked and then delegates the call to the freshly located element:

[LazyElement.java](#)

```

public class LazyElement implements WebElement, Locatable { (1)
    private final SearchContext searchContext;
    private final By locator;

    public LazyElement(SearchContext searchContext, By locator) {
        this.searchContext = searchContext;
        this.locator = locator;
    }

    private WebElement get() {
        return searchContext.findElement(locator);
    }

    @Override
    public void click() {
        get().click(); (2)
    }

    // ...

    @Override
    public Coordinates getCoordinates() {
        return ((Locatable)get()).getCoordinates(); (3)
    }
}

```

1. Implement both the `WebElement` and `Locatable` interfaces (the `Actions` class requires both of them to be able to perform actions)
2. Each time, invoke `get` to get the latest element
3. For methods from the `Locatable` interface, we need to cast the element

You can then use this in a test:

[LazyActionChainIT.java](#)

```
@Test
public void lazyActionChain() throws Exception {

    driver.get("/stale-elements.html");

    WebElement button = new LazyElement(driver, By.id("button")); (1)

    new Actions(driver)
        .click(button)
        .click(button)
        .perform();

    assertEquals("Click This Button", button.getText());
}
```

1. Create a lazy element from the driver and a locator

Using a lazy element to wrap an existing element will allow you to create long action chains. Lazy elements can be substituted in the place of any normal element and are useful for pages where an element becomes stale.

Action chains allow you to perform some complex actions that are harder, or more verbose using the methods provided directly by `WebElement`.

Advanced forms

In chapter 3 we looked at some of the basic interactions you can do with form elements. There are a couple of more advanced interactions that are useful to know about, and we'll look at them here.

Selecting multiple options

A select box usually allows you to choose only one option, but in some cases you can choose multiple options, as shown in figure [Select Boxes](#) - <http://localhost:8080/select-boxes.html>.

Select Boxes

Favourite Cat



Favourite Dogs



Figure 1. Select Boxes - <http://localhost:8080/select-boxes.html>

The HTML for the select box looks like this:

```
<select name="single" class="form-control">
    <option>Calico</option>
    <option>Ginger</option>
</select>
```

WebDriver's support library provides the `select` class to make it easier to work with select boxes:

[MultipleSelectIT.java](#)

```
Select single = new Select(driver.findElement(By.name("single")));
single.selectByVisibleText("Ginger");

assertEquals("Ginger", single.getFirstSelectedOption().getText());
```

Multiple select boxes work in much the same way. The only point to note is that you cannot get the select text directly, but you can get the option text using a loop.

Multiple select boxes have the `multiple` attribute, as per listing [Multiple select boxes](#).

Multiple select boxes

```
<select name="multiple" multiple class="form-control">
    <option>Labrador</option>
    <option>Jack Russell</option>
    <option>Sausage Dog</option>
</select>
```

If you want to select multiple options from this select list, you can do the following:

MulipleSelectIT.java

```
driver.get("http://localhost:8080/select-boxes.html");

Select multiple = new Select(driver.findElement(By.name("multiple")));

multiple.deselectAll(); (1)
multiple.selectByVisibleText("Labrador");
multiple.selectByVisibleText("Sausage Dog");

List<String> selectedOptions = new ArrayList<>();
for (WebElement option : multiple.getAllSelectedOptions()) { (2)
    selectedOptions.add(option.getText());
}

assertEquals(Arrays.asList("Labrador", "Sausage Dog"), selectedOptions);
```

1. If need be, make sure none are selected
2. Extract the text values

Selecting multiple options in a select list is not something you'll need to do often, but if you need to do it, now you know how. We'll come back to select boxes in part 3 of this book. Next, let's look at uploading files.

Uploading files

Some forms require you to upload a file, as shown in figure [File Upload - http://localhost:8080/file-upload.html](#). To do this the file needs to be transferred from the computer running the tests, to the computer running the browser (which is the same computer if you are using a local browser), and then to the server. You don't interact with the file upload using the mouse. This will result in a system pop-up appearing that you cannot dismiss. Instead, you use `sendKeys` method of `WebElement` to enter the location of the file into the browser.

File Upload

No file selected.

Figure 2. File Upload - <http://localhost:8080/file-upload.html>

File uploads can be recognized in HTML by the input type being `file`. The form is likely to have an action of `post` and may also have an `enctype`:

```
<form class="form" method="post" enctype="multipart/form-data" action="/upload.html">
    <input type="file" name="file"/>
    <input type="submit" value="Upload"/>
</form>
```

If you are running your test remotely (that is, you are using a `RemoteWebDriver`), then you need to set the file detector so that WebDriver knows to transfer the file.

WebDriver will magically copy the file from your local machine to where the browser is running!

FileUploadIT.java

```
if (driver instanceof RemoteWebDriver) { (1)
    ((RemoteWebDriver) driver).setFileDetector(new LocalFileDetector());
}

driver
    .findElement(By.name("file"))
    .sendKeys(theFile.toFile().getCanonicalPath()); (2)
```

1. If the driver is remote, you need to tell it how to find the file
2. You can just send the location of the file – you should use the canonical path

I hope this is useful to you!

Examining HTTP cookies

[1]

Om nom nom nom – cookies!!! I’m afraid that HTTP cookies [1] are not tasty snacks, though; they are in fact small pieces of information stored by the browser. They are typically used to remember users when they revisit your site, so that personalized, and often secure, content can be shown. Cookies can be restricted to just secure HTTPS requests, or only to certain domains or subdomains. While a user would not typically examine their cookies, it can be helpful to access and modify them; for example, when debugging an application.

To delete all cookies:

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

To ensure that cookies are saved:

```
// perform action that results in cookie being saved  
assertNotNull(driver.manage().getCookieNamed("cookieName"));
```

To verify that cookies are not shared across domains:

```
driver.get(firstDomainUrl);  
  
// perform operation that results in cookie being saved  
  
driver.get(secondDomainUrl);  
  
assertNull(getCookieNamed("cookieName"));
```

While WebDriver gives you access to cookies, how the web application you are testing uses them can change – it is an implementation. Tests that rely on examining the cookies are likely to be brittle. Therefore, you should only use the methods for cookies to delete them all, or when you need to debug an application.

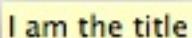
Next, we’ll look at an interesting feature of web pages – tooltips.

Tooltips

A **tooltip** is a message that appears to provide more information about how to use a control when the mouse hovers over the control for a few seconds. HTML provides a built-in mechanism to do this, the **title** attribute **tooltip**. This allows you to specify a

short line of plain text that will appear next to the element. There are also two common JavaScript mechanisms: the **tooltip** and **pop-over**.

A JavaScript tooltip is very similar to an HTML title tooltip, the primary difference being that the style can be changed (the built-in tooltip is styled by the operating system or browser). A pop-over is similar to a tooltip, but can provide more information and may have dynamic HTML content (such as links or images), and, whereas a tooltip can be expected to disappear when an element loses focus, a pop-over might stay on screen longer.



I am the title

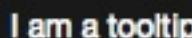
Figure 3. HTML Title Tooltip - <http://localhost:8080/tooltips.html>

The HTML for the tooltip in figure [HTML Title Tooltip](#) - <http://localhost:8080/tooltips.html> looks like the following:

```
<div title="I am the title">Title</div>
```

The text of a title-based tooltip can be extracted using `getAttribute("title")`.

```
String tip = element.getAttribute("title");
```



I am a tooltip

Figure 4. JavaScript Tooltip - <http://localhost:8080/tooltips.html>

This JavaScript tooltip in figure [JavaScript Tooltip](#) - <http://localhost:8080/tooltips.html> is an HTML `div` element:

```
<div title="I am the tooltip"
      data-toggle="tooltip" data-placement="bottom">Tooltip</div>
```

A JavaScript tooltip will require activating by moving the mouse over it. You can do this using the `Actions` class:

```
WebElement element = driver.findElement(By.id("tooltip"));

new Actions(driver).moveToElement(element).perform();

String tip = driver.findElement(By.className("tooltip-inner")).getText();
```



Figure 5. JavaScript Pop-over - <http://localhost:8080/tooltips.html>

The HTML to open the pop-over in figure [JavaScript Pop-over - http://localhost:8080/tooltips.html](#) is as follows:

```
<div title="I am the popover title"
      data-content="I am the popover content"
      data-toggle="popover" data-placement="bottom">Popover</div>
```

A JavaScript pop-over is a bit trickier. You can have more than one on the screen at a time. This means that selecting by the class name might give you the wrong result. In the case of the Bootstrap framework, you can use the attribute `aria-describedby` to find out which `div` actually contains the pop-over's content. In the following listing, the link has the value "popover535426", and you can see that this is also the ID of the `div`.

```
<a href="#" class="btn btn-default" id="popover" title=""
      data-content="I am the popover content" data-toggle="popover"
      data-placement="bottom" data-original-title="I am the popover title"
      aria-describedby="popover535426">Popover</a>

<div class="popover fade bottom in" role="tooltip"
      id="popover535426" style="top: 103px; left: 107.5px; display: block;">
      <div class="arrow" style="left: 50%; "></div>
      <h3 class="popover-title">I am the popover title</h3>
      <div class="popover-content">I am the popover content</div>
</div>
```

```
WebElement element = driver.findElement(By.id("popover"));
element.click();

String tip = driver
    .findElement(By.id(element.getAttribute("aria-describedby")))
    .findElement(By.className("popover-content"))
    .getText();
```

You can see there are a few different kinds of tooltip. To simplify accessing tooltips, let's look at introducing a helper class to extract the tooltips from a page.

Creating a tooltip extractor

There are multiple frameworks for tooltips. We would like one way to get all tooltips.

Create a `Tooltip` class that can find the tip for an element by trying different strategies. We'd like to be able to ask for the tip on an element. To do this we will need to:

1. Locate the element.
2. Determine if the tip is a tooltip or pop-over.
3. Activate the tooltip.
4. Get the tip.

You can encapsulate this as in the following class:

Tooltip.java

```
public static String tip(WebDriver driver, By by) {
    WebElement element = driver.findElement(by);

    String type = element.getAttribute("data-toggle");

    if (type == null) {
        type = "title"; (1)
    }

    switch (type) {
        case "title":
            return element.getAttribute("title");
        case "tooltip":
            new Actions(driver).moveToElement(element).perform();
            return driver.findElement(By.className("tooltip-inner")).getText();
        case "popover":
            new Actions(driver).click(element).perform();
            return driver
                .findElement(By.id(element.getAttribute("aria-describedby")))
                .findElement(By.className("popover-content"))
                .getText();
        default:
            throw new AssertionError(type);
    }
}
```

1. If this attribute is missing, then default to the "title" type

The usage is as follows:

```
String tip = Tooltip.tip(driver, By.id("tooltip"));
```

Note

Safari support

At the time of writing, the `moveToElement` method does not work in Safari.

This is another example of encapsulating behavior that may vary by implementation behind a method or interface that hides the details of whether you are talking to an HTML tooltip, a Bootstrap tooltip, or another type of tooltip.

You can now write your test in a way where you don't need to consider how the tooltip is implemented, just what the text or value of the tooltip is.

Summary

- Action chains allow you to complete complex keyboard and mouse actions.
- Working with select boxes and file upload inputs requires additional work.
- WebDriver can show you what the values of the browser's cookies are.
- There are a variety of types of tooltip, and you can encapsulate them into a utility class.

In the next chapter we will look at how to use JavaScript with web pages.

1. https://en.wikipedia.org/wiki/HTTP_cookie

Chapter 10: Executing JavaScript using the JavascriptExecutor interface

This chapter covers

- How to use the `JavascriptExecutor` interface
- Using JavaScript to get information about a web page
- Overriding built-in JavaScript to help make pages easier to test

WebDriver provides a fantastic way to access and interact with web pages in your tests. However, it can't quite do everything. Some browsers do not yet provide a complete implementation of the WebDriver API. Sometimes the WebDriver API does not provide a way to get the information that you need to test a page. In these special cases, you need to bring out the biggest gun – JavaScript.

This chapter assumes you are already comfortable with JavaScript. If you want to learn the basics of JavaScript, or you're a bit rusty, then you might wish to brush up now. There are many great resources on the Internet; for example, Codecademy [\[1\]](#) provides an online tutorial.

In this chapter, you'll see how you can use WebDriver to execute JavaScript within a browser to find out information about a page that is otherwise inaccessible using WebDriver directly. We'll cover geolocation and web notifications. You'll also learn how to override predefined JavaScript APIs, such as the Geolocation API, so you can control the behavior of the page in ways that WebDriver does not allow you to.

By the end of this chapter you'll know powerful uses of JavaScript with WebDriver that will help you test some of the tougher web pages, and be comfortable with using these techniques in your own code.

Introduction to executing JavaScript

The `JavascriptExecutor` interface allows you to invoke methods that execute JavaScript. It provides two methods: `executeScript`, which returns as soon as the script is complete, and `executeAsyncScript`, which returns once a callback function is executed (we'll talk about this shortly).

[JavascriptExecutor.java](#)

```
public interface JavascriptExecutor {
    Object executeScript(String script, Object... args);
    Object executeAsyncScript(String script, Object... args);
}
```

All the main web drivers implement this interface. To use it, you will need to cast the driver to `JavascriptExecutor`.

Typically you cast the driver to expose the `executeScript` and `executeAsyncScript` methods, and then cast the result into the class you expect:

[JavascriptExecutorUsageIT.java](#)

```
JavascriptExecutor jsExecutor = (JavascriptExecutor) driver; (1)
long windowHeight = (long) jsExecutor (2)
    .executeScript("return window.innerWidth"); (3)
```

1. Cast the driver to `JavascriptExecutor`
2. Cast the result to `long`
3. Use `return` to get the value in your test

You can pass arguments to your script for execution. Those arguments are made available as the a JavaScript array named `arguments`. The example in listing [JavascriptExecutorUsageIT.java](#) will print "Hello JavaScript!" to the JavaScript console.

[JavascriptExecutorUsageIT.java](#)

```
JavascriptExecutor jsExecutor = (JavascriptExecutor) driver; (1)
jsExecutor.executeScript(
    "var text = arguments[0];" + (2)
        "console.log(text)",
    "A string to log" (3)
);
```

1. Cast the driver.
2. You can get access to the arguments passed by your test using `arguments`
3. You pass arguments as the remaining parameters

If you want to execute methods on a `WebElement` you have already found on the page you are testing, then you can also pass that as an argument:

JavascriptExecutorUsageIT.java

```
WebElement link = driver.findElement(By.tagName("a"));

JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;
jsExecutor.executeScript("var link = arguments[0]; " +
    "link.click()", link);
```

If you are executing a piece of JavaScript that is asynchronous, such as making an a HTTP request using the `XMLHttpRequest` JavaScript class AND you want to wait for the result, you can use the `executeAsyncScript` method.

On top of the existing arguments you pass, the `arguments` variable will have extra variable appended. This variable is a JavaScript function. When you invoke `executeAsyncScript` it will wait for this function to be invoked.

As WebDriver normally expects all scripts to complete immediately, you need to set the timeout to allow enough time for your script to complete using

```
driver.manage().timeouts().setScriptTimeout .
```

JavascriptExecutorUsageIT.java

```
@Test
public void waitForAsyncJavaScript() throws Exception {

    driver.manage().timeouts()
        .setScriptTimeout(200, TimeUnit.MILLISECONDS); (1)

    JavascriptExecutor jsExecutor = (JavascriptExecutor) driver; (2)

    long startTime = System.currentTimeMillis();
    jsExecutor.executeAsyncScript(
        "var callback = arguments[arguments.length - 1];" + (3)
        "setTimeout(callback, 100);"); (4)

    assertThat(System.currentTimeMillis() - startTime, greaterThanOrEqualTo(100L)); (5)
}
```

1. Set the timeout to be longer than 100ms.
2. Cast the driver.
3. To make it easier to user, extract the callback to a variable name `callback`.

4. Invoke an asynchronous action (call the `callback` function after 100ms), which will in turn invoke the callback.
5. Check that at least 100ms has passed.

If you want to return values when executing the script, then pass them as the first argument to the callback function.

[JavascriptExecutorUsageIT.java](#)

```
String result = (String) jsExecutor.executeAsyncScript(  
    "var callback = arguments[arguments.length - 1];" + (1)  
    "setTimeout(function() {callback('Hello WebDriver!');}, 100);"; (2)  
  
assertEquals("Hello WebDriver!", result);
```

1. Invoke an asynchronous action, which will in turn invoke the callback. This time we pass "Hello WebDriver!" as the argument.

Tips

Here are some quick tips to make working with the methods of `JavascriptExecutor` smoother:

1. If you have a long piece of JavaScript, split it over multiple lines to make it easier to read.
2. Create variables for each argument. This make it easier to understand what they are for.
3. In listing [JavascriptExecutorUsageIT.java](#) you can see that the callback is within a function of the form `function() {callback(returnValue);}`. You would typically expect to see this, so if you're having problems, check if it missing.

We've had a look at the basic usage of `JavascriptExecutor`. Next, we'll look at some examples of solving problems using it.

Using JavaScript to examine a page

As the `JavascriptExecutor` has access to every element on a page, you can use it to access information that might be hard to get using normal WebDriver methods. In the next technique, we'll use it to determine if an image has been loaded.

Using JavaScript to verify that an image is loaded

You may have a page that does not load correctly, as an image is missing. This technique shows you how to use JavaScript to verify that an image is loaded.

You want to make sure that an image is correctly loaded into a page.

Cast your driver to `JavascriptExecutor` to find out if the image has loaded. A loaded image will fulfill three criteria:

1. It is marked as "complete."
2. It has a natural width, the width of the original image.
3. Its natural width is greater than zero.

These criteria give us a short JavaScript expression:

```
image.complete && typeof image.naturalWidth != 'undefined' (1)  
&& image.naturalWidth > 0
```

1. The `typeof` operator can be used to determine what type a value is

Suppose you have the page shown in figure [Images, one having failed to load - http://localhost:8080/images.html](#).



Figure 1. Images, one having failed to load - <http://localhost:8080/images.html>

You can write the following test for this page:

[VerifyingImagesIT.java](#)

```

@Before
public void setUp() throws Exception {
    driver.get("/images.html");
}

@Test
public void checkTheImagesAreLoaded() throws Exception {
    assertTrue(isImageLoaded(driver.findElement(By.id("ok"))));
    assertFalse(isImageLoaded(driver.findElement(By.id("broken"))));
}

private boolean isImageLoaded(WebElement image) {
    JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;
    return (boolean) jsExecutor.executeScript("return arguments[0].complete && " + (1)
        "typeof arguments[0].naturalWidth != 'undefined' && " +
        "arguments[0].naturalWidth > 0", image);
}

```

1. Use the JavaScript from the preceding code snippet; note that we cast to `boolean` here

Checking elements may require access to some more unusual attributes. You can find a complete list of HTML attributes on the Mozilla Developer Network:
<https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes>.

The example just shown uses JavaScript to examine the attributes. This technique can also be useful for scrolling an element into view, something you cannot do with the `WebDriver` class directly:

```

WebElement element = driver.findElement(By.id("theElement"));
((JavascriptExecutor) driver).executeScript(
    "arguments[0].scrollIntoView(true);",
    element
);

```

Accessing information that `WebDriver` does not allow you to directly is one use of `JavascriptExecutor`. Another use is to override built-in JavaScript APIs to make pages easier to test. We'll look at that in the next section.

Overriding built-in JavaScript APIs

There are a number of JavaScript APIs that are used by web pages to provide richer content. A web page can, for example, ask for the user's geographical location. However, WebDriver does not provide a way to tell the browser what location it should report, or if it should refuse to provide it. As this functionality is implemented as a JavaScript API, you can replace that API with your own implementation, which captures results and controls behavior.

Let's look at an example that uses `Date`, in figure Print The Time – <http://localhost:print-the-time.html>.

Print The Time

Print The Time

11:18:19

Figure 2. Print The Time – <http://localhost:print-the-time.html>

This page prints the current time when you click a button. We want to make sure it is correctly formatted. The time comes from the clock of the computer the browser is running on. This changes every time you click the button. Let's have a look at the page's HTML source to understand this a bit more:

[print-the-time.html](#)

```
<p><a href="javascript:printTheTime();void(0);"  
class="btn btn-default">Print The Time</a></p> (1)  
  
<p id="time"></p> (2)  
<script>  
function printTheTime() { (3)  
    var now = new Date();  
    document.getElementById("time").innerHTML  
        = now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds();  
}  
</script>
```

1. The button that prints the time
2. A place for that time to be displayed
3. A JavaScript function to print the time

The current time is found by executing `new Date()`. `Date` is a JavaScript function, and you can override JavaScript functions. If you want the page to show a specific time instead of the time on the computer (for example, 16:23:45), then you can override the `Date` function with a new function that returns a fixed, predictable value:

PrintTheTimeIT.java

```
((JavascriptExecutor) driver).executeScript(  
    "var d = new Date(2016, 1, 1, 16, 23, 45);\\n" + (1)  
    "Date = function() {return d;}" (2)  
);  
  
driver.findElement(By.linkText("Print The Time")).click();  
  
assertEquals("16:23:45", driver.findElement(By.id("time")).getText());
```

1. Create a `Date` object you want to use
2. Replace the `Date` function with one that returns your specified date/time.

This approach is great if the application uses JavaScript to determine the date. This date is controlled by the machine running your tests. If the date is determined from the application server's date (e.g. by writing the date as HTML), then you'll need to use another approach. There are a couple of options:

- Option 1 – Don't test date based pages.
- Option 2 – Have the clocks on the server and test machine synchronized, and make sure that test use the current date.
- Option 3 – Have the server modified for testing purposes so that you can tell it what time it currently is (e.g by a request parameter).

Of these three options, the third is more work, but will result in more reliable tests.

Using JavaScript to test geolocation

Some websites customize their content and behavior based on the user's location. This is known as **geolocation**.

Geolocation is a problem for testing. Like the time or random numbers, location may change based on factors outside of WebDriver's control. For example, your office may be in New York, so you write your tests assuming that you're in the New York time

zone, but they fail as soon as they are run on your Continuous Integration (CI) server – say, Jenkins – because the CI server is in London and set to UK time. You might also want to write some tests where the user is located accurately, but some tests for when the user has denied access to location information.

We've provided a page for you to experiment with at <http://localhost:8080/geolocation.html>. Opening this page will typically take you through a series of steps. First, the browser requests the user's permission to use their location (figure Geolocation requesting permission).

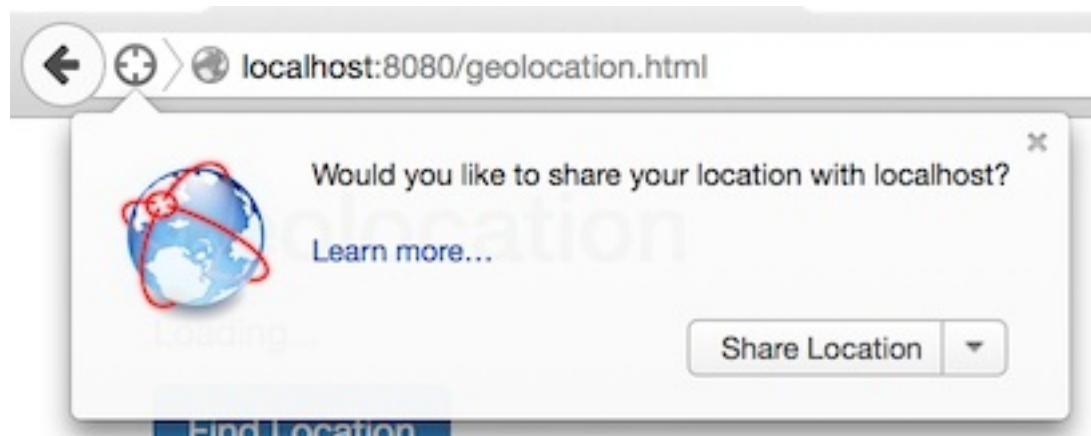


Figure 3. Geolocation requesting permission

You may see a loading message, as in figure Geolocation Loading.

Geolocation

Loading...

Find Location

Figure 4. Geolocation Loading

The page might be denied permission (figure Geolocation permission denied Error).

Geolocation

Error: 1 User denied Geolocation

Find Location

Figure 5. Geolocation permission denied Error

Or it may be granted permission, in which case you'll see the result in figure [Geolocation showing result](#).

Geolocation

You state you are at + 51.4353239, -0.1618078

[Find Location](#)

Figure 6. Geolocation showing result

The HTML for displaying the location is as follows:

[geolocation.html](#)

```
<p><a class="btn btn-primary" id="locate"
    href="javascript:locate();void(0);">Find Location</a></p> (1)
<p id="location">...</p> (2)
<script>
function locate() {
    if (!navigator.geolocation) { (3)
        $('#location').html('Not supported.')
    } else {
        $('#location').html('Loading...')
        navigator.geolocation.getCurrentPosition(function(position) {
            var c = position.coords; (4)
            $('#location').html('You state you are at + ' + c.latitude +
                ', ' + c.longitude)
        }, function(e) { (5)
            $('#location').html('Error: ' + e.code + ' ' + e.message)
        });
    }
}
</script>
```

1. A button to click to print the location
2. A place for the result to appear
3. If the browser does not support geolocation, display an error
4. If we can find the position, display it
5. Otherwise, display the error

Overriding JavaScript to control geolocation

You want to test geolocation. The location of the browser where the tests run can change, and therefore tests are unreliable.

Use `JavascriptExecutor` to replace the `navigator.geolocation.getCurrentPosition` function with your own function. The following snippet of JavaScript will do this:

```
navigator.geolocation.getCurrentPosition
  = function(ok,err){ok({coords: {"latitude": 0, "longitude": 0}});}
```

Here's how to use this in your code:

GeolocationIT.java

```
@Test
public void geoLocationInjection() throws Exception {

    JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;
    jsExecutor.executeScript(
        String.format(
            "navigator.geolocation = navigator.geolocation || {};" +
            "navigator.geolocation.getCurrentPosition = function(ok,err){" +
            "ok({coords: {latitude: %s, 'longitude': %s}});" +
            "}", (1)
            51.5106766,
            -0.1231314
        ));

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

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

    new WebDriverWait(driver, 10).until(
        (WebDriver d) -> !location.getText().equals("Loading...")); (2)

    assertEquals("You state you are at + 51.5106766, -0.1231314", location.getText());
}
```

1. Replace the geolocation function
2. Wait for the result; geolocation can take a moment to complete

This solution has one main caveat – it only works on pages where the JavaScript snippet can be executed before the location is requested. This means that if the page checks the location on startup, this technique will not work. In this case, you'll need to manually test the page.

You can also use this technique to test error scenarios, such as the user denying access to geolocation:

GeolocationIT.java

```

@Test
public void positionError() throws Exception {

    JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;
    jsExecutor.executeScript(
        String.format(
            "navigator.geolocation = navigator.geolocation || {};" +
            "navigator.geolocation.getCurrentPosition = function(ok,err){" +
            "err({'error': {" +
            "'PERMISSION_DENIED': 1, " +
            "'POSITION_UNAVAILABLE': 2, " +
            "'TIMEOUT': 3" +
            "}, 'code': %d, 'message': '%s'});",
            1,
            "User denied Geolocation"
        ));
}

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

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

new WebDriverWait(driver, 10).until(
    (WebDriver d) -> !location.getText().equals("Loading..."));

assertEquals("Error: 1 User denied Geolocation", location.getText());
}

```

These examples only implement part of the API. You can find out more about the Geolocation API here: <http://www.w3.org/TR/geolocation-API/>.

Hopefully, this section has shown you how you can use JavaScript to automate tougher page features. In the next section, we are going to look at another advanced web application feature – web notifications.

Automating web notifications

Web notifications are notifications that are pushed by the server to the browser. Rather than appearing in the browser, they appear on the user's desktop (figure [Web notification on Firefox](#)). These are also known as "toaster pop ups" or "growls."

WebDriver does not provide built-in support for these notifications. Instead, you can use the same JavaScript technique you used for geolocation.

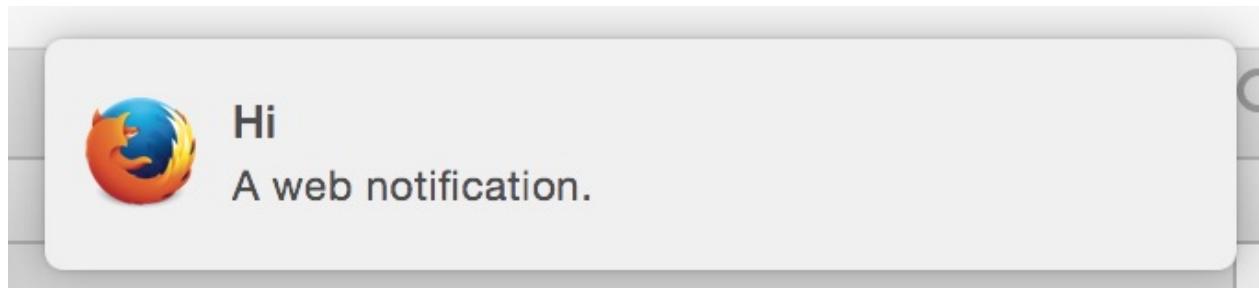


Figure 7. Web notification on Firefox

Overriding JavaScript to capture a notification

You want to verify that a web notification is displayed.

Use the `JavascriptExecutor` interface to replace the `Notification` function with an implementation that captures the result. This can be achieved by the following JavaScript snippet:

Notification.java

```
Notification = function(title, options) { (1)
    var n = this; (2)
    Notification.title = title; (3)
    Notification.options = options;
    setTimeout(function() { (4)
        (n.onshow || function() {}) (); (5)
    }, 50);
}
```

1. Replace the `Notification` function
2. Capture a reference to the newly created object (the notification)
3. Save the title and options for later inspection
4. Schedule a callback to call the `onshow` method
5. Call the `onshow` method

You can encapsulate this in an object similar to the `Select` class:

Notification.java

```

public class Notification {

    private final JavascriptExecutor javascriptExecutor;

    public Notification(JavascriptExecutor javascriptExecutor) {
        this.javascriptExecutor = javascriptExecutor;
        this.javascriptExecutor.executeScript(
            "Notification = function(title, options) { \n" +
                "    var n = this; \n" +
                "    notificationTitle = title; \n" +
                "    notificationOptions = options;\n" +
                "    setTimeout(function() { \n" +
                "        (n.onshow || function() {})(); \n" +
                "    }, 50);\n" +
            "}");
    }

    public String getTitle() {
        return (String)javascriptExecutor
            .executeScript("return notificationTitle");
    }
}

```

Now you can reuse `Notification` in as many places as you need:

[WebNotificationIT.java](#)

```

JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;
Notification notification = new Notification(jsExecutor);

driver.findElement(By.linkText("Show A Web Notification")).click();

assertEquals("A Web Notification", notification.getTitle());

```

Again, you are overriding the JavaScript function after the page has loaded. This means that if any notifications appear when it loads, you will not be able to intercept and inspect them.

This is an incomplete solution. There are also other functions, such as `onclose`, that you may want to implement. For more information about this API, see the W3C website: <http://www.w3.org/TR/notifications/>.

Summary

- The `JavascriptExecutor` interface can be used to examine elements within the page that are hard to access with WebDriver directly.
- You can override built-in JavaScript APIs to control certain web page features. This can apply to times and dates, geolocation, and web notifications.
- There's no such thing as a free lunch! If these APIs are used as soon as the page loads, you will not have time to replace them before they are first used.

In the next chapter we will look at the variety of different browsers that WebDriver supports, and how to configure them.

1. <https://www.codecademy.com/learn/javascript>

Chapter 11: What you need to know about different browsers

This chapter covers

- Features common to each browser
- Each of the main desktop browsers
- Mobile automation with Appium

When you invoke methods on the `WebDriver` interface, your computer sends a message to the web browser to tell it what to do – much in the manner a car’s driver tells the car how to act. As all browsers are different, written in different languages and having different features, this means that the `WebDriver` interface is reimplemented for each and every browser. This naturally leads to a number of different implementations, each with slight variations. When you start automating, you have to make a decision: do I automate using many different browsers, or do I focus on the one or two that are most important to my users? This is a key choice when automating, and it helps to know and understand a bit about each driver and how they work.

In this chapter, you’ll see what features are common to each browser, so you can understand what those features mean. You’ll then see how to download programs you need for your tests automatically. Browsers can leave your computer in a dirty state, or they might not be available on your computer’s operating system. You’ll see a technique to use virtual machines to overcome this. You will also see how to test iPhone and Android apps using Appium, an open source test automation framework for use with native, hybrid, and mobile web apps.

Drivers change as often as browsers are updated – and that’s pretty often. In fact, the Microsoft Edge browser had not even been released when we started writing this chapter.

You will have learned about all the major desktop, headless, and mobile browsers by the end of the chapter. This will help you save time when making choices about what browser to use with your tests.

Comparing drivers

There are a number of key features and aspects that are worth bearing in mind when dealing with any of the drivers and browsers. Here is a high-level view of them:

Browser

Naturally, if you want to test with a browser, then you'll need to install it. If you need a specific version, then you'll need to locate and download that version from the vendor's website.

Driver binary/extension

Some (but not all) drivers require you to set up or install some additional software. This is often in the form of a **driver binary**. You may need to download this and install it before you get started. If it's not a binary, it may be a **browser extension**. Whenever the browser updates, you may find that you also need to update your WebDriver client libraries (for example, by updating their versions in Maven). But don't confuse the driver binary (only needed when using the browser with WebDriver) with the **browser binary**, part of the browser itself.

Web browser engine

Each browser uses a **web browser engine** to render each web page. This affects how the page looks and behaves. Different browsers that use the same engine will behave similarly. Is it worthwhile repeating a test in two browsers that are very similar to one another?

Customization – profiles, preferences, and options

Many drivers allow you to customize the browser in some way, such as setting whether or not user history is saved. This can be useful in some circumstances, but you'll probably find that you generally leave the settings alone. If you need to customize your browser to run your tests, then think about which browsers you want to run your tests on.

Extensions

Some browsers allow you to install extensions or plugins. If the application you are testing makes a lot of use of specific extensions, you can install them. But generally you'll find that the time it takes to set up the extensions might be better spent documenting and executing a manual test.

Handling self-signed certificates and protected domains

When developing a new application, it is not uncommon to use a **self-signed certificate** for secure HTTPS connections. An application may require a **client certificate** too. Getting rid of this pop up can be a bit tricky, so we'll cover it in this chapter.

Synthetic versus native events

A driver can use **native events**, **synthetic events**, or a mixture of the two. A synthetic event is one that is implemented using JavaScript (similar to using the `JavascriptExecutor` interface). JavaScript can do things like click buttons that are disabled, which means that the behavior might not accurately reflect what would happen when a user clicks the button. A native event is one that more accurately mimics the user's interaction with the browser. Pretty much all drivers support native events, so you'll probably never need to consider synthetic events, but there may be very rare scenarios where you need to use them.

Note

Local vs. remote drivers

A **local driver** is one that is running on a machine you are working on. This has the benefits of allowing you to set up, control, and debug your tests easily. It has the disadvantage that a badly managed driver can crash your computer, or it might upgrade itself midway through your test. A **remote driver** runs on a remote machine. It has the advantages of being well behaved and managed by someone else, but the disadvantages of being harder to access, control, and debug.

As many mature test suites run using remote drivers on a Selenium grid, we try to make sure all techniques in the book run on both.

You can refer to appendix A for more information about setting up a grid.

Table [Driver summary](#) lists the platform, engine, and type of events supported by each driver.

Table 1. Driver summary

Name	Platform	Engine	Events	Notes
Firefox	Desktop	Gecko	Native	
Chrome	Desktop	Blink	Native	Separate binary download, mobile emulation mode
Internet Explorer	Desktop	Trident	Native	Windows only
Microsoft Edge	Desktop	EdgeHTML	Native	Windows only
Safari	Desktop	WebKit	Native and synthetic	Use Appium for mobile testing
HtmlUnit	Desktop	Java	Native	Headless
PhantomJS	Desktop	WebKit	Native	Headless
Appium	Mobile	Various	Native	Requires Xcode and/or Android SDK

Self-signed certificates

When you are developing a new website, you may use a self-signed certificate for HTTPS requests. This typically results in a pop up that you cannot dismiss (see figure [Firefox showing a pop up about an untrusted connection that must be dismissed](#)).

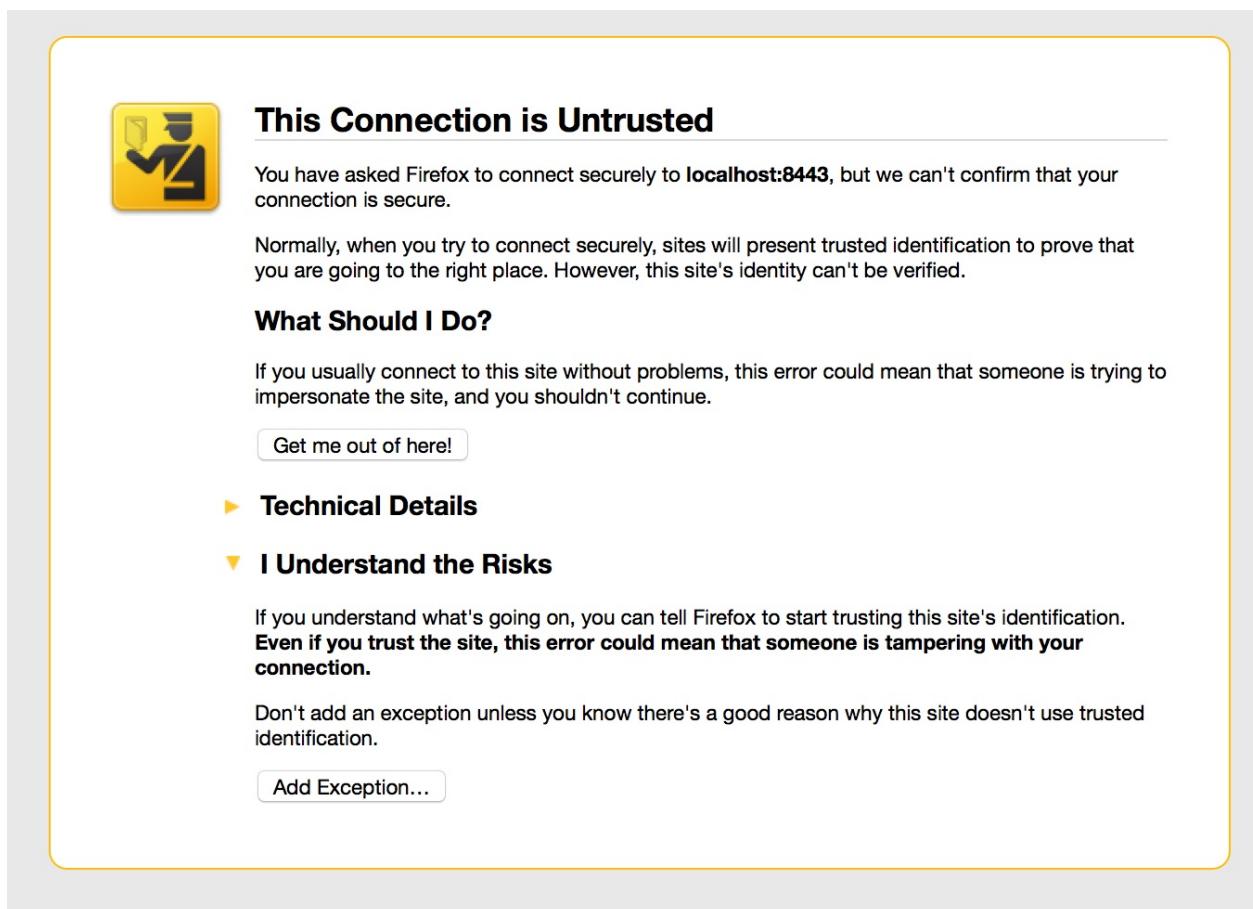


Figure 1. Firefox showing a pop up about an untrusted connection that must be dismissed

To prevent this appearing, set the following capability:

```
capabilities.setCapability(CapabilityType.ACCEPT_SSL_CERTS, true);
```

Desktop browsers

Not so long ago, most people browsing the Internet were using Internet Explorer. But this has changed dramatically.

In 2005, Internet Explorer was top dog with 85% market share.^[1] In 2010, Firefox was the head honcho with 45% share, and in 2015 Chrome dominated with 65%. Things change rapidly, but I would not write off either Firefox or IE just yet. For one thing, plenty of governmental and commercial systems still run only on IE. For another, Firefox is the easiest browser to use with WebDriver.

Browser usage also varies by region. While Chrome dominates in most countries, Opera is popular in Africa, UC Browser in India and China, Firefox in the Africa and the Far East, and IE in Japan.

You'll notice we don't cover every single different browser in this chapter. Some browsers are much more popular, and their drivers much better supported, than others. Also, browsers often share engines, and by testing using one browser you can have some confidence that your web application works in other browsers that use the same or a similar engine.

Firefox



Figure 2. Firefox

`FirefoxDriver` is probably your first choice for a driver. It runs **Mozilla Firefox** on Windows, OS X, and Linux. It supports all the driver features you'll need and requires the least amount of setup. `FirefoxDriver` uses the **Gecko** engine and supports extensions.

You can customize it using the `FirefoxProfile` class. With this class you can control how SSL certificates are handled (using the `setAcceptUntrustedCertificates` and `setAssumeUntrustedCertificateIssuer` methods). If you want to install and automate multiple versions, you can do so using the `FirefoxBinary` class.

To use Firefox, add the following to your pom.xml file (note you'll need to set `selenium.version` in your properties):

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-firefox-driver</artifactId>
  <version>${selenium.version}</version>
  <scope>test</scope>
</dependency>
```

Chrome

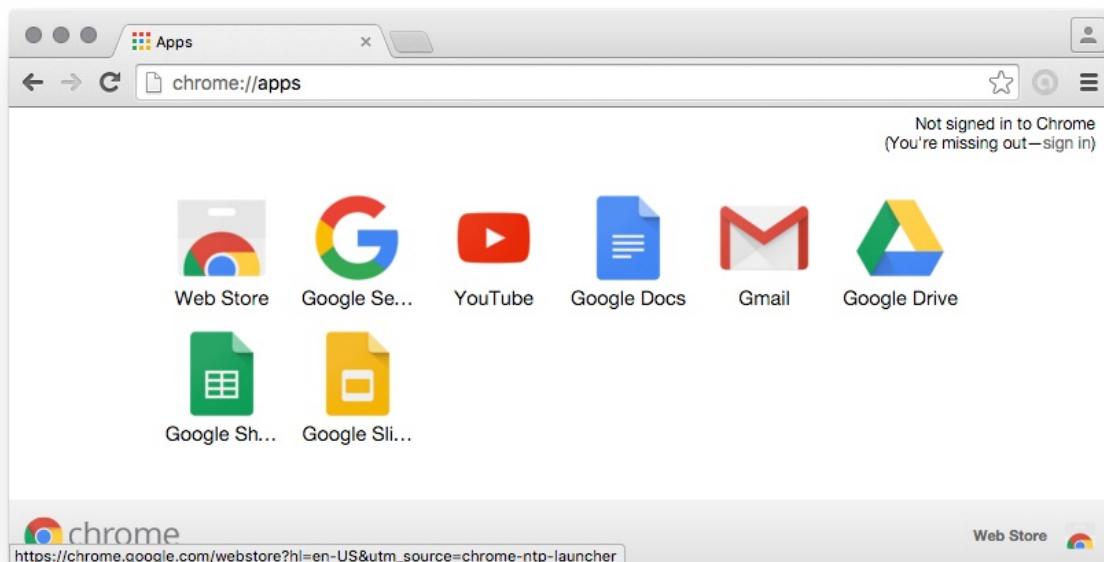


Figure 3. Chrome

`ChromeDriver` runs the **Google Chrome** browser on Windows, OS X, and Linux. You need to download a driver binary to use it. It supports extensions. `ChromeDriver` uses the **Blink** engine, also used by recent versions of Opera. This means you might be confident enough using this driver that you do not need to test Opera. Blink is a fork of the **WebKit** engine, which is used by Safari and PhantomJS.

You can customize `ChromeDriver` using the `ChromeOptions` class.

To use Chrome, you'll need to add the following to your pom.xml file:

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-chrome-driver</artifactId>
  <version>${selenium.version}</version>
  <scope>test</scope>
</dependency>
```

To use `ChromeDriver`, you must tell it where to find a special file known as the **chromedriver binary**. This driver can be downloaded from <https://sites.google.com/a/chromium.org/chromedriver/downloads>. You'll need to download and unpack the correct version for your operating system. Once you have done that, you need to set the system property `webdriver.chrome.driver` to the path of the binary:

```
System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
```

You can save the binary anywhere, and one good place is within your project (for example, in `src/test/bin`). On OS X and Linux you'll need to make it executable – you can do this from the terminal:

```
chmod +x chromedriver
```

Automatically downloading the driver binary

Chromedriver binaries are large, and require you to download different versions for different operating systems. This technique shows you how to automatically download the correct version.

You want to use `ChromeDriver` without having to manually download the binary.

Automatically download the correct driver binary for your computer's operating system. As you may want to do this in a different way for different drivers, you can use the **strategy pattern** by creating an interface that can be implemented for each browser.

[WebDriverBinarySupplier.java](#)

```
public interface WebDriverBinarySupplier {  
    /**  
     * @param target A directory to place the binary in.  
     * @return The name of the binary file.  
     */  
    Path get(Path target) throws IOException;  
}
```

There are four basic steps that you need to do to get a binary:

1. Determine which URL to get it from for your computer's operating system.
2. Download the file.
3. Unpack or unzip the file.
4. Make sure it is executable.

We implement these steps in the `ChromeDriverBinarySupplier` class. To make it easy to understand, we define some variables first:

[ChromeDriverBinarySupplier.java](#)

```
private Logger LOGGER = getLogger(ChromeDriverBinarySupplier.class);  
private String BASE_PATH = "http://chromedriver.storage.googleapis.com";  
private Path download = Paths.get(getProperty("java.io.tmpdir"),  
    "chrome-driver"); (1)  
private String osName = getProperty("os.name"); (2)  
private String osArch = getProperty("os.arch");  
private String os = osName.contains("win") ? "win" :  
    osName.contains("nix") ? "linux" : "mac";  
private int arch = os.matches("linux|mac") && osArch.endsWith("64") ? 64 : 32; (3)
```

1. The name and download destination
2. You typically need to know what operating system and architecture you are running on to get the binary
3. Chrome has a 64-bit version for Linux and Mac, a 32-bit version for everyone else

We then implement those four steps in this method:

[ChromeDriverBinarySupplier.java](#)

```

@Override
public Path get(Path driverDir) throws IOException {

    Path driverPath = resolvePath(driverDir);
    if (!driverPath.toFile().exists()) { (1)
        if (!download.toFile().exists()) {
            download();
        }
        unzipFiles(driverPath);
        makeExecutable(driverPath);
    }
    return driverPath;
}

```

1. Do not download again if you already have it

Some helper methods used in that method are listed here:

[ChromeDriverBinarySupplier.java](#)

```

private void download() throws IOException {
    try (FileOutputStream fos = new FileOutputStream(download.toFile())) {
        fos.getChannel().transferFrom(createChannel(), 0, Long.MAX_VALUE);
    }
}

private void unzipFiles(final Path driverPath) throws IOException {
    LOGGER.info("extracting chrome driver to " + driverPath);

    try (FileSystem fileSystem = createFile()) {
        walkFileTree(fileSystem.getPath("/"), new SimpleFileVisitor<Path>() {
            @Override
            public FileVisitResult visitFile(
                Path file, BasicFileAttributes attrs) throws IOException {

                LOGGER.info("unzipping " + file); (1)
                copy(file, driverPath, REPLACE_EXISTING);

                return CONTINUE;
            }
        });
    }
}

private void makeExecutable(Path path) {
    LOGGER.info("making " + path + " executable");

    if (!path.toFile().setExecutable(true)) { (2)

```

```

        throw new IllegalStateException("failed to make " + path +
            " executable");
    }
}

private ReadableByteChannel createChannel() throws IOException {
    URL url = createUrl();

    LOGGER.info("downloading " + url + " to " + download);
    return newChannel(url.openStream()); (3)
}

private FileSystem createFile() throws IOException {
    return newFileSystem(create("jar:file:" + download), emptyMap()); (4)
}

private URL createUrl() throws IOException {
    return new URL(BASE_PATH + "/" + lastRelease() +
        "/chromedriver_" + os + arch + ".zip");
}

private Path resolvePath(Path driverDir) {
    return driverDir.resolve(os.equals("win") ?
        "chromedriver.exe" : "chromedriver");
}

private String lastRelease() throws IOException {

    URL url = new URL(BASE_PATH + "/LATEST_RELEASE");

    try (Scanner scanner = new Scanner(url.openStream())) {
        return scanner.useDelimiter("\n").next().trim(); (5)
    }
}

```

1. We assume we will only find a single file in the zip
2. Finally, we need to make it executable
3. Download the file using Java 7's NIO APIs
4. Java 7 can unpack a jar file automatically – a jar file is actually a zip file, so we take advantage of that here
5. Get the latest version from this URL

The preceding example has some code for downloading and unzipping files. This could be extracted into a utility class for reuse in other drivers.

We discussed downloading the driver binary manually (which will require you to update it each time the browser is updated), and automatically downloading the latest version. If you are using Maven, there is another option, the **Selenium driver-binary-downloader-maven-plugin**.^[2] This is a Maven plugin that downloads driver binaries automatically. To use it, you need to add a few lines to your pom.xml file:

pom.xml

```
<plugins>
  ...
  <plugin>
    <groupId>com.lazerycode.selenium</groupId>
    <artifactId>driver-binary-downloader-maven-plugin</artifactId>
    <version>1.0.7</version>
    <executions>
      <execution>
        <phase>pre-integration-test</phase> (1)
        <goals>
          <goal>selenium</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  ...
</plugins>
```

1. Adding a `pre-integration-test` phase makes sure this is executed before the integration tests are run

You should see something similar to the following in the output:

```
[INFO] Setting maven property - ${phantomjs.binary.path} =
selenium_standalone/osx/phantomjs/64bit/phantomjs
[INFO] Setting maven property - ${webdriver.chrome.driver} =
selenium_standalone/osx/googlechrome/64bit/chromedriver
[INFO] Setting maven property - ${webdriver.opera.driver} =
selenium_standalone/osx/operachromium/64bit/operadriver
```

All the binaries you might need will be downloaded. For example, running on OS X creates the following files:

```
selenium_standalone/osx/googlechrome/64bit/chromedriver
selenium_standalone/osx/operachromium/64bit/operadriver
selenium_standalone/osx/phantomjs/64bit/phantomjs
```

Chrome has one feature that might make it the preferred choice of browser to test with: the ability to emulate mobile devices. The next technique looks at how to do that.

Mobile testing using Chrome's mobile emulation

Chrome provides the ability to emulate different mobile devices. This can be useful for early identification of problems with your site on mobile devices, without the time and effort required to set up a full mobile automation system.

You want to test for mobile, but without too much special setup.

Use Chrome's mobile emulation mode.

If Chrome knows about the device you want to emulate, you can **specify a known mobile device**.

To get Chrome to emulate a mobile device, you need to know the name of the device you want to emulate. To find this out, open the Dev Tools by right-clicking on the page and choosing Inspect. Click the small Toggle Device Mode button shown in figure [Toggle device mode button](#). This button looks like a small iPad.

Toggle device mode button

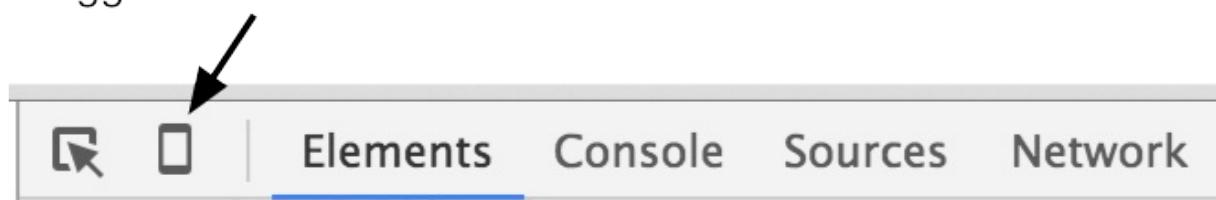


Figure 4. Toggle device mode button

The new toolbar shown in figure [Device mode toolbar](#) will appear, allowing you to choose the device you want.

Choose device (e.g.
iOS or Android)

Device orientation

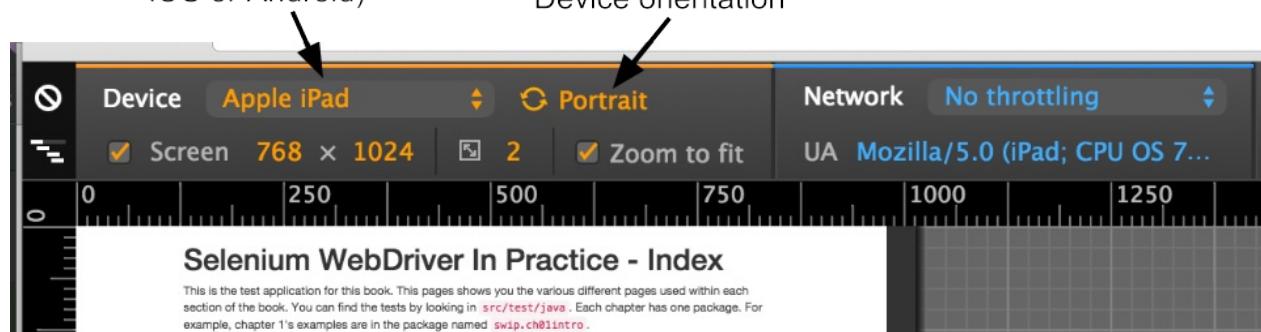


Figure 5. Device mode toolbar

You need the exact string shown – in this example, "Apple iPad".

To tell the `ChromeDriver` class to start in mobile emulation mode, you must pass some properties of `DesiredCapabilities`:

- `chromeOptions` – Must be set to a map
- `chromeOptions.mobileEmulation` – Must be set to a map too
- `chromeOptions.mobileEmulation.deviceName` – Must be set to the device name

For example:

[ChromeMobileEmulationKnownDeviceIT.java](#)

```
mobileEmulation.put("deviceName", "Apple iPad");

Map<String, Object> chromeOptions = new HashMap<>();

chromeOptions.put("mobileEmulation", mobileEmulation);

DesiredCapabilities capabilities = DesiredCapabilities.chrome();

capabilities.setCapability(ChromeOptions.CAPABILITY, chromeOptions);

driver = new ChromeDriver(capabilities);
```

If Chrome does not know about the device, you must **specify individual device attributes**. You need to specify:

- `chromeOptions.mobileEmulation.deviceMetrics.width` – The width of the device's screen
- `chromeOptions.mobileEmulation.deviceMetrics.height` – The height of the device's screen
- `chromeOptions.mobileEmulation.deviceMetrics.pixelRatio` – The pixel ratio
- `chromeOptions.mobileEmulation.userAgent` – The value of the HTTP user agent header

The following example uses the same values as the Apple iPad:

[ChromeMobileEmulationDeviceAttributesIT.java](#)

```
Map<String, Object> deviceMetrics = new HashMap<>();  
  
deviceMetrics.put("width", 768);  
deviceMetrics.put("height", 1024);  
deviceMetrics.put("pixelRatio", 2);  
  
Map<String, Object> mobileEmulation = new HashMap<>();  
  
mobileEmulation.put("deviceMetrics", deviceMetrics);  
mobileEmulation.put("userAgent", "Mozilla/5.0 (iPad; CPU OS 7_0 like Mac OS X) AppleWebKit/537.51.1 (KHTML, like Gecko) Version/7.0 Mobile/11A465 Safari/9537.53");  
  
Map<String, Object> chromeOptions = new HashMap<>();  
  
chromeOptions.put("mobileEmulation", mobileEmulation);  
  
DesiredCapabilities capabilities = DesiredCapabilities.chrome();  
  
capabilities.setCapability(ChromeOptions.CAPABILITY, chromeOptions);
```

You can run the test application in mobile mode if you like. See <https://github.com/selenium.webdriver-book/source/> for details.

Chrome's mobile emulation feature is a great and inexpensive way to test mobile devices. It doesn't require any extra software installed on either developers' PCs or your CI system, so there is very little cost in getting set up. Naturally, there are a few differences between emulation and testing on a real device:

- A desktop machine is likely to be much more powerful than a mobile device, so performance will greatly differ.
- Many mobile features (such as the camera) are not supported.

If your site uses mobile features and you want to emulate them, then you might want to consider Appium, which we'll talk about shortly.

Internet Explorer

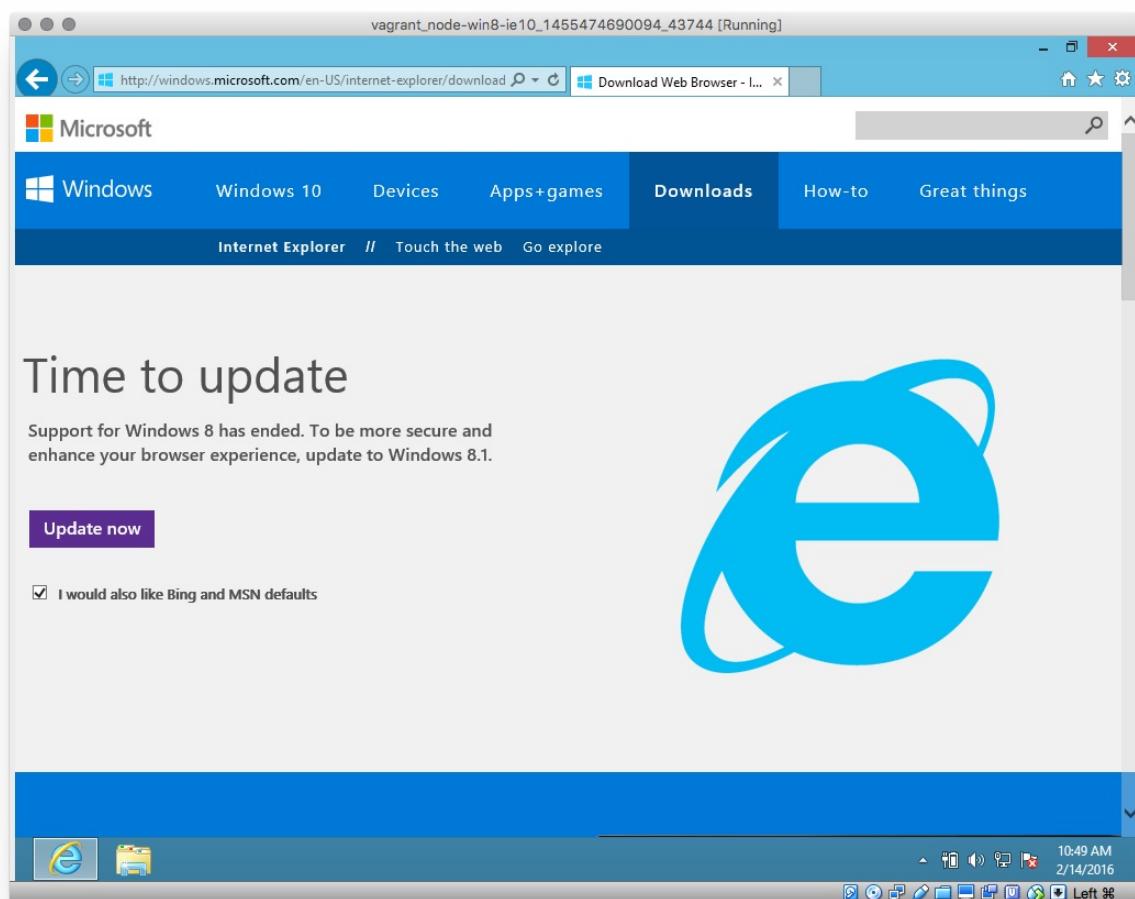


Figure 6. Internet Explorer

`InternetExplorerDriver` runs the **Internet Explorer** browser on Windows. It requires you to download a driver binary to use. It supports extensions and uses the **Trident** engine.

To use Internet Explorer, add the following to your pom.xml file:

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-ie-driver</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
</dependency>
```

Using a Vagrant virtual machine to run IE

You may want to use a browser that is not available on your operating system, such as Internet Explorer when your computer is an Apple Mac. This technique shows you how to create a virtual machine to run a different operating system, allowing you to test a greater variety of browsers.

Your desktop machine is not running Windows, but you want to run your tests on Internet Explorer. Or, you want to start from scratch each time with a working machine, as your tests can make the machine dirty.

Use VirtualBox and Vagrant to create a virtual machine (VM) that you roll back to a known good state before running your tests.

VirtualBox (figure [VirtualBox](#)) allows you to run a virtual computer (a "virtual box" or "virtual machine") within your desktop computer. This is what allows you to run another operating system without needing another piece of computer hardware to run it on.

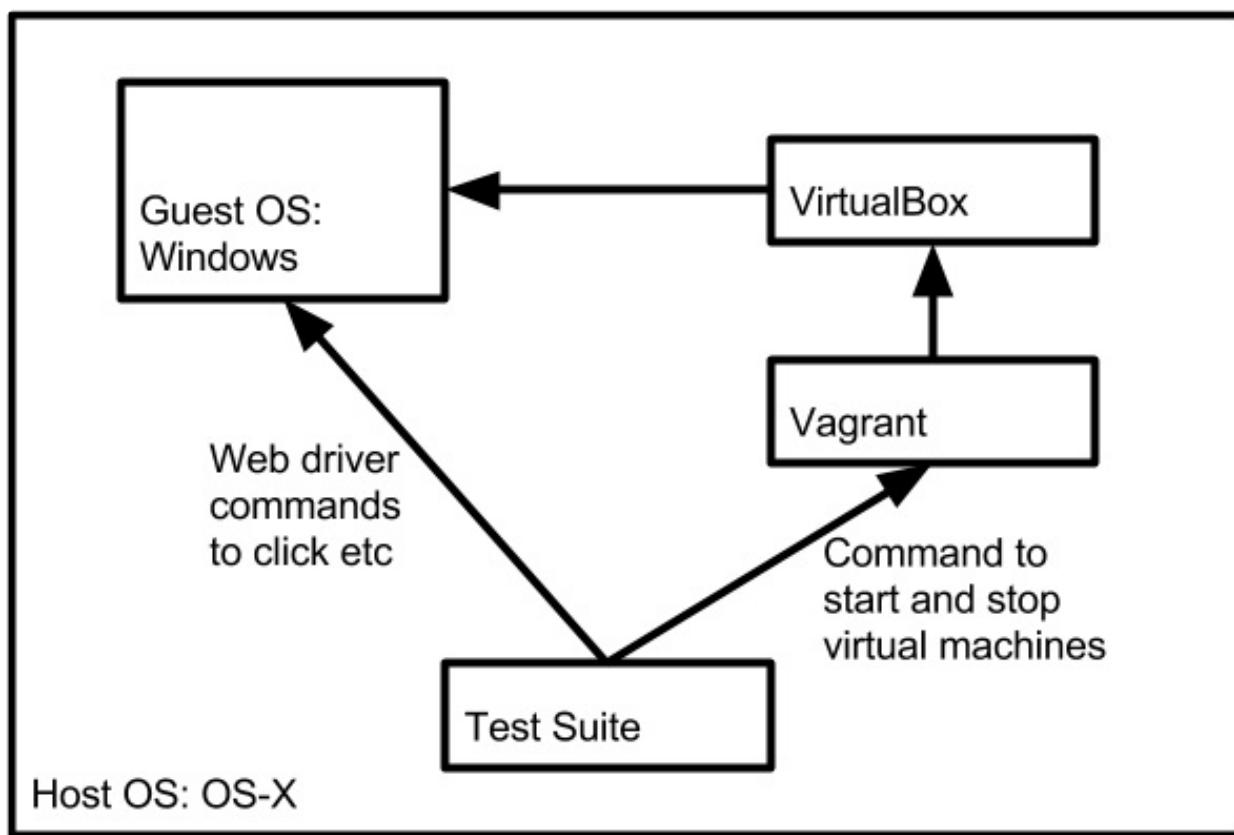


Figure 7. VirtualBox

Vagrant is a tool used to control VirtualBox, telling it how to create the virtual machine.

How to find out more about VirtualBox and Vagrant

If you've not used either VirtualBox or Vagrant, we recommend you try out the tutorials on their websites, as it will make this technique much easier to understand:

- <https://www.virtualbox.org>
- <https://www.vagrantup.com>

As is often the case with Windows, this will require a series of manual steps.

You are going to create a Windows 8 virtual machine with Internet Explorer 10 installed on it. First, some prerequisites. You need to install VirtualBox and Vagrant.

What is the difference between a "guest" and a "host" OS?

The **host OS** is the operating system running on your desktop machine. The operating system running on the virtual machine is called the **guest OS**.

Make sure you have an appropriate license from Microsoft for running Windows.

Create an empty directory (we called ours win8-ie10), and in that directory create a file named Vagrantfile. It should contain the following content:

Vagrantfile

```
Vagrant.configure("2") do |config|  
  
  config.vm.box = "win8-ie10" (1)  
  config.vm.box_url = "http://aka.ms/vagrant-win8-ie10" (2)  
  config.vm.communicator = "winrm" (3)  
  config.vm.network "forwarded_port", guest: 4444, host: 14444 (4)  
  
  config.vm.provider "virtualbox" do |v|  
    v.gui = true (5)  
  end  
  
end
```

1. A name for your virtual box
2. The URL to download the virtual box from
3. Enable Windows Remote Management
4. Make sure that port 14444 on your desktop is forwarded to port 4444 on your virtual machine – this is the port that the IE driver listens on
5. Turn on the user interface so you can interact with it.

What is a "port forward"?

VirtualBox allows you to map a port on your host machine to one on your guest machine. This is called a **port forward**. It means that when the port on your host is accessed, data is forwarded to the guest. This allows you to access services and programs listening on a port on the guest machine as if they were running on the host.

Next, type `vagrant up`. After a few minutes, you should see Windows running within a window on your desktop. On the Windows guest machine, download and install Java. Now is a good time to check that IE is working and can connect to websites. Use IE to download the latest versions of `IEDriverServer` (this runs a server that allows WebDriver to speak to IE) and `selenium-server-standalone-2.52.0.jar` (this runs the WebDriver server) from <http://docs.seleniumhq.org/download/>. Save these on the machine's desktop. Unzip the zip file containing `IEDriverServer.exe` onto the desktop.

You need to start up the standalone driver now. To do this, open the Windows `cmd` application (for example, by pressing Windows+R and then typing `cmd`). Enter the following:

```
cd c:\Users\IEUser\Desktop
"c:\Program Files\Java\jre1.8.0_73\java.exe" -Dwebdriver.ie.driver=IEDriverServer.exe \
-jar selenium-server-standalone-2.52.0.jar -port 4444
```

You may need to change the version from `2.52.0`. You should see the output in figure [Expected output from command](#).



The screenshot shows a Windows Command Prompt window titled "Windows PowerShell" with the path "C:\Windows\WinSxS\x86_microsoft-windows-commandprompt_31bf3856ad364...". The window contains the following text:

```
C:\Windows\WinSxS\x86_microsoft-windows-commandprompt_31bf3856ad364...
The system cannot find message text for message number 0x2350 in the message file for Application.
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Windows\system32>cd c:\Users\IEUser\Desktop

c:\Users\IEUser\Desktop>"c:\Program Files\Java\jre1.8.0_73\bin\java.exe" -Dwebdriver.ie.driver=IEDriverServer.exe -jar selenium-server-standalone-2.52.0.jar -port 4444
11:17:24.080 INFO - Launching a standalone Selenium Server
11:17:24.652 INFO - Java: Oracle Corporation 25.73-b02
11:17:24.652 INFO - OS: Windows 8 6.2 x86
11:17:24.700 INFO - v2.52.0, with Core v2.52.0. Built from revision 4c2593c
11:17:24.866 INFO - Driver class not found: com.opera.core.systems.OperaDriver
11:17:24.866 INFO - Driver provider com.opera.core.systems.OperaDriver is not registered
11:17:24.878 INFO - Driver provider org.openqa.selenium.safari.SafariDriver registration is skipped:
registration capabilities Capabilities [{browserName=safari, version=, platform=MAC}] does not match the current platform WIN8
11:17:25.915 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:4444/wd/hub
11:17:25.915 INFO - Selenium Server is up and running
```

Figure 8. Expected output from command

Now, you can create a test!

[VagrantInternetExplorerIT.java](#)

```
private WebDriver driver;

@Before
public void setUp() throws Exception {
    driver = new RemoteWebDriver(
        new URL("http://localhost:14444/wd/hub"),
        DesiredCapabilities.internetExplorer() (1)
    );
}

@Test
public void openGoogle() throws Exception {
    driver.get("http://www.google.com");
}
```

1. Note the port number is the one we set up earlier

Using `InternetExplorerDriver` is the only way to test on Internet Explorer with WebDriver. This example allows you to set up a Windows machine that lives inside your machine. Before you run your first test, you can save the virtual machine. To do this, you need to install a plugin into Vagrant:

```
vagrant plugin install vagrant-vbox-snapshot
```

You can then take a snapshot of the machine:

```
vagrant snapshot take win-ie
```

And each time, you can start fresh using the machine when the snapshot was taken:

```
vagrant snapshot go win-ie
```

You can find a full list of virtual machines on [Microsoft's website](#).

As mentioned earlier, Internet Explorer may require some manual setup, so you may also want to do the following before you use the snapshot:

- Disable "protected mode" so you can test browser plugins.
- Enable deletion of browser history.
- Turn off the pop up blocker.

- Disable autocomplete of passwords.
- Disable certificate warnings.

These changes reduce the chances of a pop up interrupting your tests, causing them to fail.

VirtualBox and Vagrant allow you to create multiple guest VMs on a single host OS.

Microsoft Edge

Microsoft Edge is a new browser (as of 2015) from Microsoft that supports the WebDriver protocol. You need to install Microsoft WebDriver to use it. There isn't a complete implementation of the WebDriver protocol at the time of writing, so we won't talk much about it in case it changes. Edge uses the **EdgeHTML** engine. This is a fork of Internet Explorer's Trident engine, so you may expect to see similar behavior.

Safari

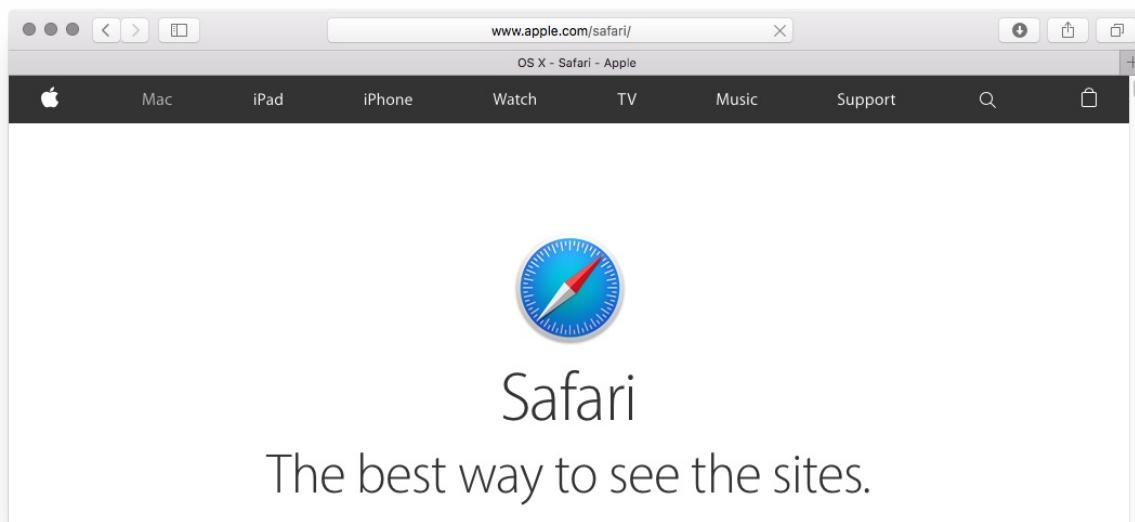


Figure 9. Safari

`SafariDriver` is the driver for the **Safari** browser on OS X. It uses both injected JavaScript and native events to control the browser. To use it, you need to install an extension. `SafariDriver` does not automate some features, such as alerts, file uploads, or drag and drop. As Safari is based on the WebKit engine, it behaves in a similar way to Chrome.

To use Safari, add the following to your pom.xml file:

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-safari-driver</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
</dependency>
```

Now that we have looked at the main desktop browsers WebDriver supports, we'll look at a special class of browser – the headless browser.

Headless browsers

One thing you might want to consider is headless testing. A headless browser is one that does not display a user interface – it's missing its "head"! This allows a computer to access web pages, but without the overhead of rendering the graphics to the screen.

Headless browsers tend to be faster, as they do not have the overhead of rendering graphics. However, this can make problems with tests harder to diagnose, as you cannot simply view the web page at the point your test fails.

Let's take a look at three ways to run a headless browser.

HtmlUnit

`HtmlUnitDriver` is the driver for HtmlUnit. This is not a browser in the normal sense; it's a Java implementation of a browser. This means that it'll run on any operating system, even one without a browser installed. It uses a different JavaScript engine to other browsers, which means that JavaScript does not always work as expected. We've found that pages that rely on jQuery may not work. This means it might not be a great choice for a JavaScript-rich website. But it is very fast, so it may be an appropriate choice for some simple websites.

You cannot capture the screen when something goes wrong. This means that diagnosing failures might take longer than with other drivers.

To use HtmlUnit, add the following to your pom.xml file:

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-htmlunit-driver</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
</dependency>
```

Note that the driver's version does not always seem to be up to date.

PhantomJS

PhantomJS is a headless browser based on the WebKit engine. This means it is similar to Chrome and Safari. It is headless, but it does not have some of the drawbacks of HtmlUnit; for example, JavaScript works as expected. It can be a good choice for using on your continuous integration server. But you should be aware that the small differences to Chrome can result in hard-to-diagnose bugs in your tests.

To run PhantomJS on your computer, you need to install it. If you are on OS X and you have Brew installed, then run:

```
brew install phantomjs
```

If you are running Linux, then you can use your package manager to install it. On Windows, you can download it from [the PhantomJS website](#).

You need to add the following to your pom.xml file to get the `PhantomJSDriver` class:

```
<dependency>
    <groupId>com.codeborne</groupId>
    <artifactId>phantomjsdriver</artifactId>
    <version>1.2.1</version>
    <exclusions>
        <exclusion>
            <groupId>org.seleniumhq.selenium</groupId> (1)
            <artifactId>selenium-java</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-remote-driver</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

1. We exclude these dependencies, as they are often out of date

We've had trouble testing JavaScript pop ups with PhantomJS, so watch out for that.

If you want to run a browser without a head, another option is to use XVFB. We'll look at that next.

XVFB: The X11 virtual frame buffer

X11-based operating systems, such as Linux, can create a virtual screen to run tests using the `xvfb` command. This is installed by default in many Linux versions. To use it, you only need to do the following in your terminal:

```
Xvfb :99 &
export DISPLAY=:99
```

Any graphical process that starts in that terminal from now on will start without a GUI. This gives you some of the benefits of running headless, with the benefits of having a browser that is well supported.

iPhone, iPad, and Android – Appium

Mobile browsing is set to overtake desktop browsing globally any time now – in fact, maybe even between when I write these words and when you read them!

Mobile drivers have some common features:

- Typically you need to install the phone's SDK. This can limit you to the platform that the OS vendor favors.
- You can often test various versions of the operating system by providing a capability to the driver.
- They are typically remote drivers that you need to start up before you use them.

A popular way to automate mobile browsers is using Appium.^[3] Not only does it support multiple browsers, but it can allow you to test native applications. It's also well supported by cloud testing solutions such as Sauce Labs.

You need to install some prerequisites to use Appium: Xcode^[4] if you want to test on iOS, and the Android SDK^[5] if you want to test on Android. Once you've installed the Android SDK, you'll also need to download an emulator. As this process can change, I

won't document it here. The instructions can be found on [Appium's website](#).

The Appium user interface (figure [Appium toolbar](#)) provides a number of useful features.

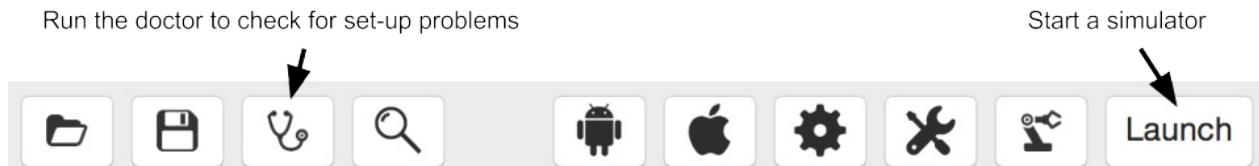


Figure 10. Appium toolbar

It's worthwhile running the Appium Doctor to check everything is working:

```
Running iOS Checks
✓ Xcode is installed at /Applications/Xcode.app/Contents/Developer
✓ Xcode Command Line Tools are installed.
✓ DevToolsSecurity is enabled.
✓ The Authorization DB is set up properly.
✓ Node binary found at /usr/local/bin/node
✓ iOS Checks were successful.

Running Android Checks
✓ ANDROID_HOME is set to "/usr/local/opt/android-sdk"
✓ JAVA_HOME is set to "/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/."
✓ ADB exists at /usr/local/opt/android-sdk/platform-tools/adb
✓ Android exists at /usr/local/opt/android-sdk/tools/android
✓ Emulator exists at /usr/local/opt/android-sdk/tools/emulator
✓ Android Checks were successful.

✓ All Checks were successful
```

Appium and Appium Doctor can be run from the command line, which is ideal for a CI server:

```
$ appium-doctor
Running iOS Checks
...
$ appium
info: Welcome to Appium v1.4.7 (REV 3b1a3b3ddffa1b74ce39015a7a6d46a55028e32c)
info: Appium REST http interface listener started on 0.0.0.0:4723
info: Console LogLevel: debug
```

It's well worth making sure you can start the device you want to test on before you start writing your tests. This can be done in Xcode, or within Android Studio.

Finally, you use `AppiumDriver` in a similar manner to other remote drivers:

AppiumPhoneIT.java

```

public class AppiumPhoneIT {
    private AppiumDriver<MobileElement> driver; (1)

    @Before
    public void setUp() throws Exception {
        DesiredCapabilities capabilities = new DesiredCapabilities();

        capabilities.setCapability(MobileCapabilityType.DEVICE_NAME, "iPhone 5"); (2)
        capabilities.setCapability(MobileCapabilityType.PLATFORM_NAME, "iOS");
        capabilities.setCapability(MobileCapabilityType.PLATFORM_VERSION, "9.2");
        capabilities.setCapability(CapabilityType.BROWSER_NAME, "safari"); (3)

        driver = new IOSDriver<>(
            new URL("http://127.0.0.1:4723/wd/hub"),
            capabilities
        );
    }

    @After
    public void tearDown() throws Exception {
        driver.quit();
    }

    @Test
    public void openGoogle() throws Exception {
        driver.get("http://www.google.com");

        MobileElement element = driver.findElement(By.xpath("//button"));
        element.click();
    }
}

```

1. The `AppiumDriver` class exposes the `MobileElement` class, which has more methods than `WebElement`
2. The device you need – don't worry, Appium will log a list of devices if you don't know which one you need
3. You need either a `browserName` or an `app` name: this should be "safari"

You'll see the device start up, the browser load, and then your page appear as per figure [Appium iOS flow](#).

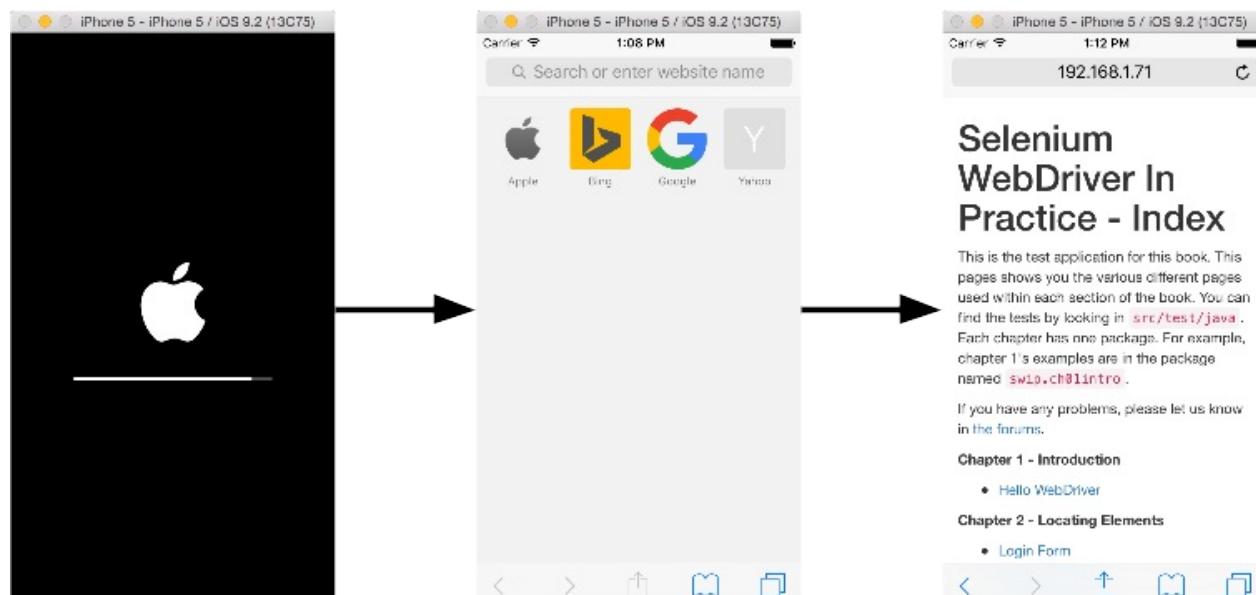


Figure 11. Appium iOS flow

The code for Android is very similar, with just a change to the parameters:

AppiumAndroid.java

```
capabilities.setCapability(MobileCapabilityType.DEVICE_NAME, "Nexus");
capabilities.setCapability(MobileCapabilityType.PLATFORM_NAME, "Android");
capabilities.setCapability(MobileCapabilityType.PLATFORM_VERSION, "4.4");
capabilities.setCapability(MobileCapabilityType.BROWSER_NAME, "browser");
```

This will result in you seeing the browser loading and your page appearing as per figure [Appium Android flow](#).

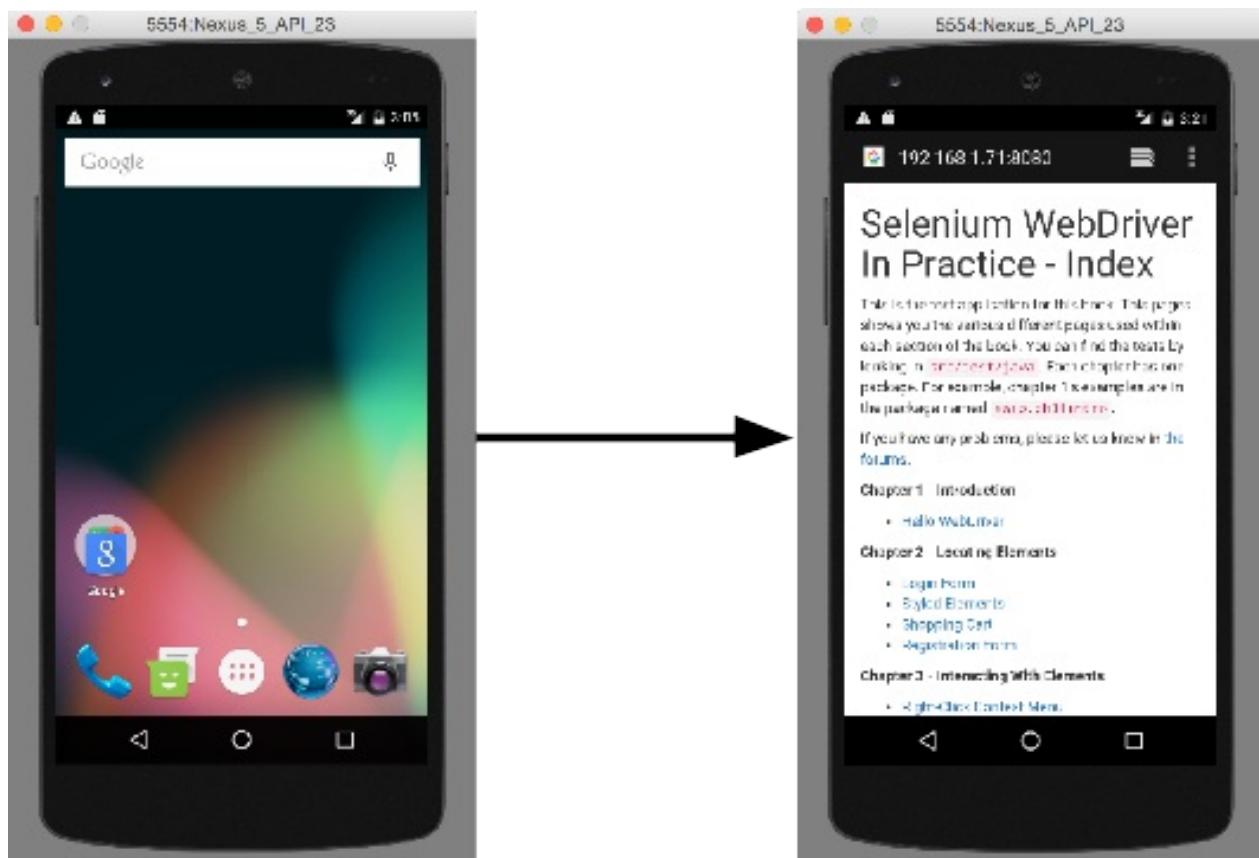


Figure 12. Appium Android flow

A couple of notes on Appium. First, it only supports a subset of locators: XPath and class name. Also, some methods, such as swipe and pinch, are not currently supported. But it is rapidly improving, so by the time you read this, this might have changed!

Summary

- Not all browsers support all features. Use table [Driver summary](#) to compare them.
- Some browsers need manual configuration before you can use them. This is different for each browser.
- Some browsers are similar to others, so using just one of them to test can give you a lot of value.
- VirtualBox and Vagrant will allow you to use browsers that are not native to your OS.
- Appium is your one-stop-shop for mobile browser automation. You can use this for both iPhone and Android testing.

In the next chapter we will look at ways you can decorate WebDriver to add to or modify its behavior.

1. https://en.wikipedia.org/wiki/Usage_share_of_web_browsers
2. <https://github.com/Ardesco/selenium-standalone-server-plugin>
3. <http://appium.io/>
4. <https://developer.apple.com/xcode/>
5. <https://developer.android.com/sdk/installing/index.html>

Chapter 12: Wrapping WebDriver and WebElement

In this chapter

- Using the `EventFiringWebDriver` web driver to capture screen-shots whenever a page loads
- Decorating web driver to allow you to use URL relative to base URL
- Using a HTTP proxy server to capture HTTP status codes
- Wrapping `WebElement` for HTML table with a `Table` interface
- Wrapping `By` locators inside enum constants

Wrapper an object can be used to add new behavior, or modify existing behavior. This is known as the **decorator pattern** [1]. Wrapping web driver allows you add useful functionality to your automation code. For example, you could log each URL as it is opened, or close pop-ups that interrupt your tests. As you wrap the web driver, you can use the wrapped WebDriver in all your existing code, and get access to new features you add by wrapping it everywhere you use web driver, but still having all the features of the wrapped driver.

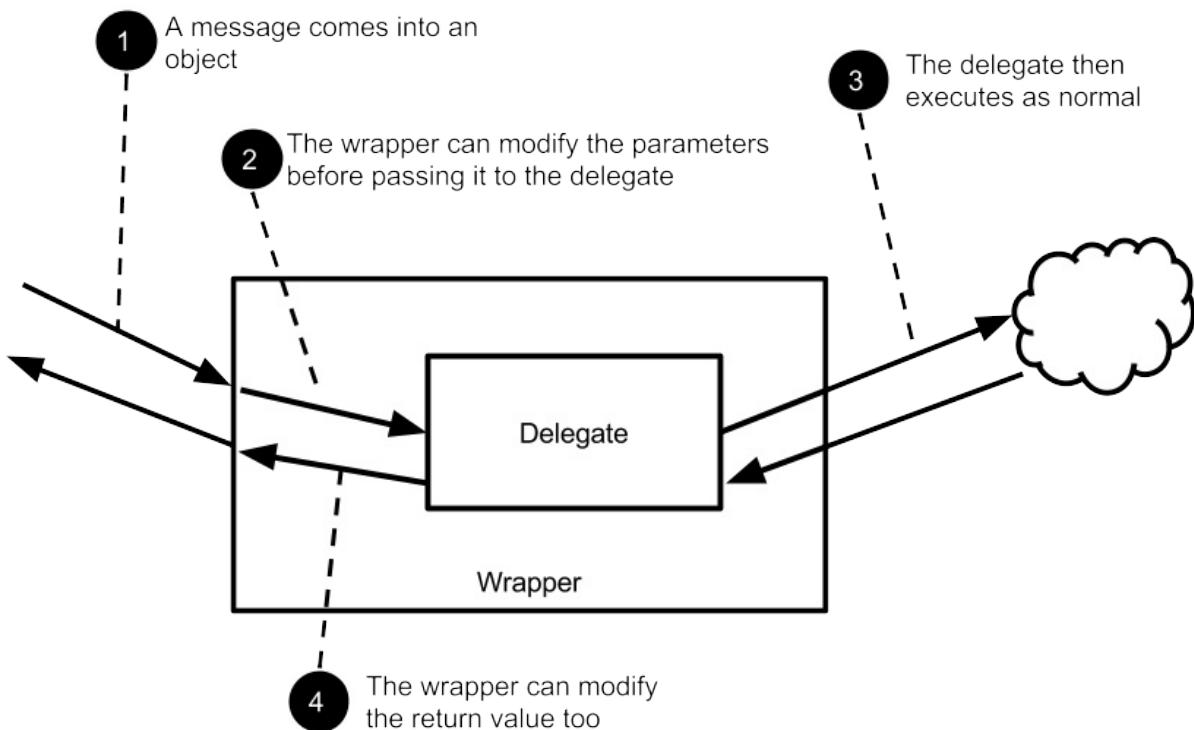


Figure 1. Wrapping WebDriver

By wrapping the driver, can add various different abilities to the driver without changing any existing code. This means that you can save a great deal of time if you decide they would be useful after you've started writing your test suite. For example:

- Log each URL opened, and each method invoked.
- Record whenever the page's title or headers change.
- Capture an image if an exception occurs.
- Capture an image after a method is invoked to create an animation of the browser.
- Automatically recognize HTTP 5xx and 4xx errors.
- Automatically log the user in.
- Closing pop-ups that might interrupt your code.

In this chapter, you will see an existing wrapper, the event firing web driver and see how to use it to capture screen-shots. You'll then see an example of modifying the behavior of web driver's `get` method so that it can take URLs relative to a base URL. Finally, you see how to deal with very tough problem – capturing HTTP status codes.

By the end of the chapter you will have learned all the skills you need to wrap drivers.

How do I find existing examples of wrapped drivers?

Drivers that decorate a web driver typically implement the `WrapsDriver` interface. If you're implementing your own wrapper you may wish to do this too.

Event firing web driver

The standard Selenium Java library already contains a wrapper, the **event firing web driver**. This wraps an existing driver to provide a method that allows you to register listeners which are invoked when certain actions (such as pages being loaded) are performed.

Taking screen-shots each time a page is loaded

The Event firing web driver allows you to write code that listens to requests sent to a normal web driver and take action based on them. This technique shows you how to take a screen shot each time a page is loaded.

You want to take a screen-shot when pages are loaded to make it easier to trace problems in your code.

Wrap your driver in an `EventFiringWebDriver` and register a listener. To do this, you need to create an object of the sub-class `WebDriverEventListener`. This class has a large number of methods, so to avoid having to implement them all, you can sub-class `AbstractWebDriverEventListener`. This has empty implementations of each of the methods, so you only need to implement the ones you want.

ScreenshotIT.java

```
EventFiringWebDriver driver = new EventFiringWebDriver(this.driver); (1)

driver.register(new AbstractWebDriverEventListener() {

    @Override
    public void afterNavigateTo(String url, WebDriver driver) {
        File screenshotFile = ((TakesScreenshot) driver)
            .getScreenshotAs(OutputType.FILE); (2)

        System.out.println("saved " + url + " as " + screenshotFile);
    }
});

driver.get("/index.html"); (3)
```

1. Wrap the existing driver.
2. Take a screen-shot and save it to a file.
3. This will navigate to a page, and therefore it will take a screenshot.

This shows an example of using the event firing web driver to wrap a web driver, and then save a screen-shot. The event firing web driver allows you to perform operations before and after a number of actions:

- Navigation to a URL, or forwards and backwards in history.
- Finding an element.
- Clicking.
- Entering text.
- Running a script.

It also allows you to perform additional actions when there is an exception using the `onException` method, though it doesn't allow you to perform any remedial actions should an exception occur.

`EventFiringWebDriver` is perfect for tasks such as taking screenshots, or logging web driver's behavior. Unless you throw an exception, `EventFiringWebDriver` does not allow you to change how your web driver behaves. You cannot change the arguments passed to your web driver, or change the value returned. To do this, we need to do something a bit more complex.

`EventFiringWebDriver` uses JDK dynamic proxies under the hood. We'll encounter them in more detail next.

Base URL driver

In a chapter 7 we looked at a technique that allowed you to use base URL for your URLs in your test by injecting the URL into your test class. To use that technique you need to modify every class that you wanted to use it in. This can be achieved by wrapping a web driver to automatically prefix the base URL if it is missing very time you invoke `get`.

Creating a driver that automatically prefixes URLs with a base URL

Injecting base URLs into tests is verbose, we want to have a driver that automatically prefixes a base URL.

Create a method that returns a web driver that wraps an existing web driver, but prefixes the URLs with a base URL.

The main item we need to achieve this is a driver that delegates all its calls to another driver, use the **delegate pattern** [2] to delegate the code to another class. We can then sub-class that driver to override methods with new implementations.

DelegatingWebDriver.java

```
class DelegatingWebDriver implements WebDriver {  
    private final WebDriver driver;  
  
    DelegatingWebDriver(WebDriver driver) {    (1)  
        this.driver = driver;  
    }  
  
    @Override  
    public Options manage() {  
        return driver.manage();  (2)  
    }  
  
    @Override  
    public void get(String url) {  
        driver.get(url);  
    }  
  
    ...  
}
```

1. DelegatingWebDriver wraps WebDriver
2. It only delegates the call to WebDriver

This class doesn't do very much on it's own, but lets look at what a sub-class of it can do.

BaseUrlDriver.java

```

class BaseUrlDriver extends DelegatingWebDriver {
    private final URI baseUrl;

    BaseUrlDriver(WebDriver driver, URI baseUrl) {
        super(driver);
        this.baseUrl = baseUrl;
    }

    @Override
    public void get(String url) {
        super.get(!url.contains(":/") ? baseUrl + url : url); (1)
    }
}

```

1. It has some logic to use the url directly if the url has "://" in it, otherwise append it to `baseUrl` and call the `get` method in `DelegatingWebDriver`

This class overrides the `get` method. If the URL passed to is absolute (e.g. <http://localhost:8080/index.html>) then is passed to the delegate, but if it is relative (.e.g /index.html) then the base URI is prefixed to it. This can be used in place of any driver. It's convenient if the base URL can be changed using a system property:

```

new BaseUrlDriver(new FirefoxDriver(),
    URI.create(System.getProperty("webdriver.baseUrl")));

```

Finally, you can use it in your code:

```
driver.get("/index.html");
```

And set it to different locations when you run your tests:

```
mvn ... -Dwebdriver.baseUrl="http://test-server"
```

The above example has a very specific use: to allow you to use the same tests to test applications on different hosts. To make this completely versatile, we need to implement all the interfaces that the `RemoteWebDriver` class implements. As `WebDriver` does not implement other useful interfaces such as `JavascriptExecutor`, you'll need to cast them as shown in listing [DelegatingWebDriver.java](#).

[DelegatingWebDriver.java](#)

```

class DelegatingWebDriver
    implements WebDriver, JavascriptExecutor, TakesScreenshot, (1)
    HasInputDevices, HasCapabilities {

    ...

    @Override
    public Object executeAsyncScript(String script, Object... args) {
        return ((JavascriptExecutor) driver).executeAsyncScript(script, args); (2)
    }

    @Override
    public <X> X getScreenshotAs(OutputType<X> target)
    throws WebDriverException {
        return ((TakesScreenshot) driver).getScreenshotAs(target); (3)
    }
    ...
}

```

1. Implement these useful interfaces to make the `DelegatingWebDriver` more versatile
2. Cast `driver` to `JavascriptExecutor` to execute its `executeAsyncScript` method
3. Cast `driver` to `TakesScreenshot` to execute its `getScreenshotAs` method

Those Finds Interfaces

`RemoteWebDriver` also implements some other interfaces, `FindsById`, `FindsByClassName` and so on. Each of them provides 2 finders methods so there are 16 methods all together. Do we need to implement these interfaces to make the `DelegatingWebDriver` compatible with Selenium library? The answer is no. If we implemented those interfaces, there would be 16 more methods in `DelegatingWebDriver` which not only would make it difficult to use, but also encourage people to write lengthy tests as well.

```

@Override
public WebElement findElementByXPath(String using) {
    return ((FindsByXPath) delegate).findElementByXPath(using);
}

@Override
public List<WebElement> findElementsByXPath(String using) {
    return ((FindsByXPath) driver).findElementsByXPath(using);
}

```

And we don't think those interfaces are necessary. Not only they are not necessary, but they are harmful as well. They would bloat the API and make `DelegatingWebDriver` hard to use.

On the contrary, `SearchContext` is a much better design, it only has two methods but these two methods can be used to handle all kinds of `Finds`, `By.ById`, `By.ByClassName` and so on.

```
public interface SearchContext {  
    List<WebElement> findElements(By by);  
    WebElement findElement(By by);  
}
```

So we haven't actually implemented those interfaces. We don't encourage you to implement them either. Try to use `SearchContext` instead. But we are going to provide another interface to replace `SearchContext` as well. You will see that in next chapter.

You can also add new methods to your wrapped driver:

```
public void click(By by) {  
    findElement(by).click();  
}
```

We will cover this in Chapter 15 when we notice chained call makes the source call too long and we add this method to shorten the call. And we'll look at more ways to build your framework in part three of the book. In the next section you will see more complex examples where we add new functionality to a driver.

How to find out a page's HTTP status code

A **HTTP Status Code** is a number the server returns to the browser to indicate the status of the request. Common codes are:

- 200 OK
- 403 Unauthorized
- 404 Page Not Found
- 500 Internal Server Error
- 503 Server Temporarily Unavailable

These code are useful, as they can be used to quickly identify errors in pages, before anything else happens. However, WebDriver does not make the status code easily available.

Creating a WebDriver wrapper that captures HTTP status codes using a HTTP proxy

We'd like to be able to see the last HTTP status code of the last page loaded.

Use a HTTP proxy that captures the status codes, and expose those codes as a Spring bean in your tests.

To do this, we'll start a HTTP proxy server as part of our tests. We'll connect the driver to it, so that whenever the driver requests a page, we capture information about the page. We'll extend the Spring configuration from chapter 7 to do this. If you want a reminder about how this works, now is a good time to refer back to it.

1. The test framework will start a proxy server
2. When the test framework starts the driver and browser, it'll also tell the browser to use the proxy server
3. When requests from the browser are sent, they will be routed via the proxy server
4. When the framework starts the test, it'll inject both the driver and an object that reports the HTTP status code `HttpStatusCodeSupplier`
5. The proxy server will capture the HTTP status code so that the test script can see what the status code was

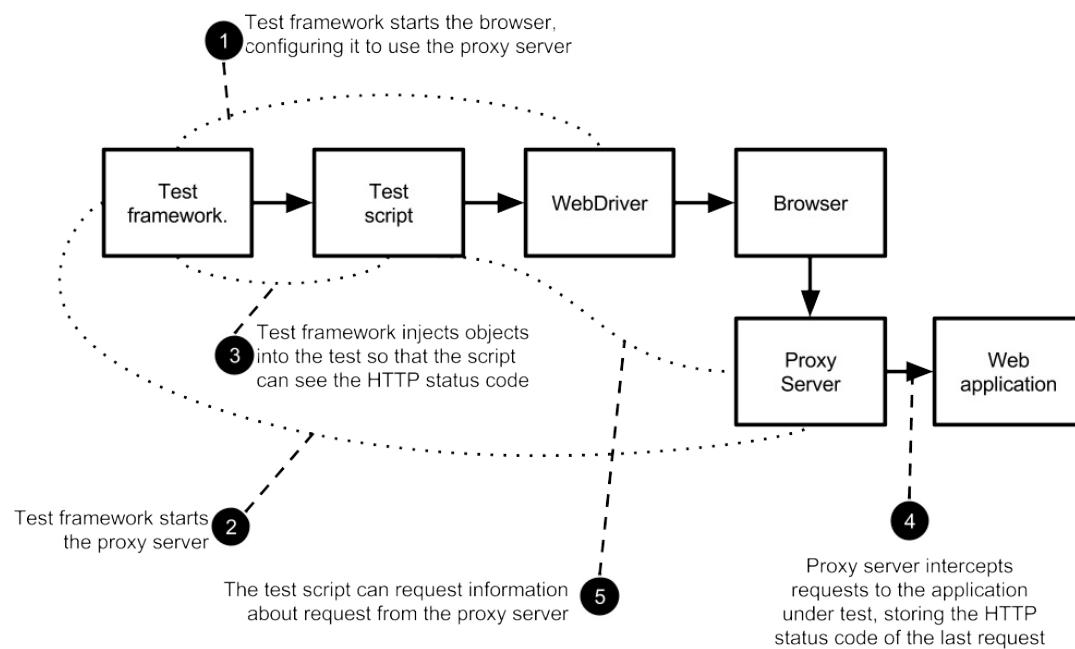


Figure 2. Using A Proxy Server

Rather than write our own HTTP proxy server, we'll use [Lightbody Proxy](#) [3]. This is a proxy that can capture the HTTP traffic of tests. Add this to your `pom.xml`:

pom.xml

```
<dependency>
    <groupId>org.littleshoot</groupId>
    <artifactId>littleproxy</artifactId>
    <version>1.1.0-beta2</version>
    <exclusions>
        <exclusion>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

When we start our tests, we'll need to start the proxy server. This can be done by modifying the `WebDriverConfig` and adding a few new Spring beans to it:

WebDriverConfig.java

```

private static int freePort() throws IOException { (1)
    try (ServerSocket serverSocket = new ServerSocket(0)) {
        return serverSocket.getLocalPort();
    }
}

@Bean(destroyMethod = "abort") (2)
public HttpProxyServer proxyServer(HttpFiltersSource httpFiltersSource)
    throws IOException, InterruptedException {
    InetSocketAddress inetSocketAddress =
        new InetSocketAddress(InetAddress.getLocalHost(), 0);
    return DefaultHttpProxyServer.bootstrap()
        .withNetworkInterface(inetSocketAddress)
        .withFiltersSource(httpFiltersSource)
        .withPort(freePort())
        .start();
}

```

1. This method returns a port that is not in use currently.
2. We must stop the server, Spring will not know how to do this automatically, so we must specify the names of the methods that Spring must call. This will make sure that the proxy server is shutdown at the end of the tests.

Add add this method to add the `PROXY` as a capability to `WebDriver`:

[WebDriverConfig.java](#)

```

@Bean
public DesiredCapabilities desiredCapabilities(HttpProxyServer proxyServer) {
    DesiredCapabilities capabilities =
        new DesiredCapabilities("firefox", "", Platform.ANY);
    // ...
    String httpProxy = proxyServer.getListenAddress().toString().substring(1); (2)
    Proxy proxy = new Proxy().setHttpProxy(httpProxy).setSslProxy(httpProxy)
        .setFtpProxy(httpProxy).setSocksProxy(httpProxy);
    capabilities.setCapability(CapabilityType.PROXY, proxy); (1)
    // ...
    return capabilities;
}

```

1. Set the `PROXY` capability to tell WebDriver to use the proxy.
2. Remove a leading "/"

Next you'll need a class to capture the last status code, as well as allow your test to access them. This class must implement the `HttpFiltersSource` interface.

HttpStatusCodeSupplier.java

```

public class HttpStatusCodeSupplier extends HttpFiltersSourceAdapter { (1)

    private int httpStatusCode;

    @Override
    public HttpFilters filterRequest(HttpServletRequest originalRequest) {
        return new HttpFiltersAdapter(originalRequest) {
            public String uri;

            @Override
            public HttpResponse proxyToServerRequest(HttpObject httpObject) {
                if (httpObject instanceof HttpServletRequest) {
                    HttpServletRequest httpRequest = (HttpServletRequest) httpObject;
                    uri = httpRequest.getUri(); (2)
                }
                return super.proxyToServerRequest(httpObject);
            }

            @Override
            public HttpObject serverToProxyResponse(HttpObject httpObject) {
                if (httpObject instanceof HttpServletResponse) {
                    HttpServletResponse httpResponse = (HttpServletResponse) httpObject;
                    if (uri.endsWith("html")) { (3)
                        httpStatusCode = httpResponse.getStatus().code(); (4)
                    }
                }
                return super.serverToProxyResponse(httpObject);
            }
        };
    }

    public int get() throws InterruptedException {
        if (httpStatusCode == 0) { (5)
            throw new IllegalStateException(
                "no request has yet been successfully intercepted");
        }
        return httpStatusCode;
    }
}

```

1. `HttpFiltersSourceAdapter` implements the `HttpFiltersSource` interface. Extending it reduces the amount of code needed.
2. Save the URL for later.

3. Only intercept pages that are HTML, assuming you don't want status codes for JavaScript or CSS resources.
4. Save the status code in a field.
5. If the status code is zero, no code has been captured yet.

This will need to be added to the `WebDriverConfig` too:

```
@Bean
public HttpStatusCodeSupplier httpStatusCodeSupplier() {
    return new HttpStatusCodeSupplier();
}
```

Finally, you can use this in your code as per listing [HttpStatusCodeIT.java](#) below.

[HttpStatusCodeIT.java](#)

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = WebDriverConfig.class)
public class HttpStatusCodeIT {

    @Inject
    private WebDriver driver;
    @Inject
    private HttpStatusCodeSupplier httpStatusCodeSupplier;

    @Test
    public void notFound() throws Exception {
        driver.get("/not-found.html");

        assertEquals(404, httpStatusCodeSupplier.get());
    }

    @Test
    public void resourceNotFound() throws Exception {
        driver.get("/resource-not-found.html");

        assertEquals(200, httpStatusCodeSupplier.get());
    }
}
```

This technique may not work if you are running your tests on Selenium Grid. This will be because the grid will need to connect to the proxy, running on your desktop PC, and your office network might not allow this. If you use a grid, then you probably want to test this first to make sure you can get any changes made necessary.

This approach could be extended to expose more information about the request. For example, how long it takes to complete, or the full HTTP request headers.

Wrapping WebElement for HTML table with a Table interface

Not only we can wrap `WebDriver` interface, we can wrap `WebElement` interface as well.

In Chapter 2, we introduced a technique to locate cell element on a table using CSS or Xpath. But CSS and XPath locators are difficult to write and understand by many people. So we are going to introduce a new technique to allow people to locate table cell using its row and column number. And for functional tests purpose, both row and column number start from 1, so it make it easy to communicate with non technical people on the project.

For example, in the table cell, we will use (row number, column number) to indicate the position of the cell on the table,

Table 1. Header number, (row number, column number) of a table

Header 1	Header 2	Header 3
(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)
(4,1)	(4,2)	(4,3)

Then we are going to introduce a technique to locate the contents on the table cell using its row number and column number.

Encapsulating table access Within a Table class

While tables are made up from rows, columns, headings, cells, you may find you want to access them by the cell. Lets look at a technique to do this.

You regularly have to create complex table locators and it is proving to be time-consuming to write, and difficult to maintain.

Encapsulate table access within a `Table` object. HTML tables have a standard layout, they have a head, and a body. Both contain cells, in the case of the body, using the `td` HTML tag.

Let's have a look at the methods you would want a table class to provide:

Table.java

```
public interface Table extends WebElement {  
  
    WebElement getBodyCell(int rowNumber, int columnNumber);  
  
    int getWidth();  
  
    int getBodyHeight();  
}
```

We won't cover all the methods (`getWidth` and `getBodyHeight` are left as exercises to the user). Instead, we will focus in on the methods that gets cells. You may notice that the `Table` class extends the `WebElement` class; this means that you can use this object wherever you previously used a table's `WebElement`. You can drop it into existing code easily, and without sacrificing functionality.

You are going to create a simple implementation that is passed the table element from the page via the constructor as follows:

SimpleTable.java

```
public class SimpleTable implements Table {  
    private final WebElement tableElement;  
  
    public SimpleTable(WebElement tableElement) {  
        this.tableElement = tableElement;  
    }  
}
```

You would use this as follows:

TableIT.java

```
Table table = new SimpleTable(driver.findElement(By.id("users-table")));
```

The methods from `SearchContext` are directly delegated to the table (the *decorator pattern*):

```
@Override  
public WebElement findElement(By by) {  
    return tableElement.findElement(by);  
}  
  
@Override  
public List<WebElement> findElements(By by) {  
    return tableElement.findElements(by);  
}  
  
...
```

The most interesting method is `WebElement getBodyCell(int rowNumber, int columnNumber)`. This method will need to find a cell by its row and column numbers. You can use the `TdBy` to do this.

[SimpleTable.java](#)

```
@Override  
public WebElement getBodyCell(int rowNumber, int columnNumber) {  
    return tableElement  
        .findElement(By.tagName("tbody"))  
        .findElement(TdBy.cellLocation(rowNumber, columnNumber));  
}
```

You can build on this class to provide some more interesting and useful methods. For example, you can encapsulate the strategy of finding the correct column number:

[ColumnNumberFinder.java](#)

```

class ColumnNumberFinder {

    private final SearchContext context;

    ColumnNumberFinder(SearchContext context) {
        this.context = context;
    }

    public int find(String headerText) {
        for (int columnNumber = 1; ; columnNumber++) {
            if (context
                .findElement(By.cssSelector(String.format("th:nth-child(%d)", columnNumber
)))
                .getText().equals(headerText)) {
                return columnNumber;
            }
        }
    }
}

```

This is an example of the **strategy pattern** [4]. It is a strategy for finding out something about the page. Let's look at an example of applying this strategy:

SimpleTable.java

```

public class SimpleTable extends DelegatingWebElement implements Table {

    private final ColumnNumberFinder columnNumberFinder;

    public SimpleTable(WebElement delegate) {
        super(delegate);
        columnNumberFinder = new ColumnNumberFinder(
            delegate.findElement(By.tagName("thead"))); (1)
    }

    @Override
    public WebElement getBodyCell(int rowNum, String header) {
        int columnNumber = columnNumberFinder.find(header); (2)
        return getBodyCell(rowNum, columnNumber);
    }

    ...
}

```

1. Create an object to find the column.
2. Find the column.

More on table

The technique about table here is to assist you to locate certain element on table when you don't care about the content of entire table. If you need to validate the contents on the table, this technique is not sufficient for that purpose. We will cover comprehensive validation of table in Chapter 16.

Wrapping By locators inside enum constants

Some pages can be very complex, and this means the XPaths and CSS selectors you have to use end up being very long. For example, on the shopping cart page (shown in figure [Shopping cart http://localhost:8080/shopping-cart.html](http://localhost:8080/shopping-cart.html)), all the inputs have very long names:

Shopping Cart

Item	Unit Price	Qty	Cost
Selenium WebDriver Book (Softbound + eBook edition) remove	\$44.99	1	\$44.99

Purchase

Figure 3. Shopping cart <http://localhost:8080/shopping-cart.html>

```
<input type="text"
      name="cartDS.shoppingcart_ROW0_m_orderItemVector_ROW0_m_quantity"
      class="form-control input-sm" value="1" size="2"/>
```

It would require the element locating code to be written like this:

A **By** locator with a really long name

```
webDriver.findElement(
    By.name("cartDS.shoppingcart_ROW0_m_orderItemVector_ROW0_m_quantity"))
```

And it is not only this one element that has a long name, all of the elements are that long. You need a way to organize those locators to make the code less verbose. One way is to introduce a constant like `QUANTITY`:

```
private static final By QUANTITY
    = By.name("cartDS.shoppingcart_ROW0_m_orderItemVector_ROW0_m_quantity");
```

So the code becomes:

```
webDriver.findElement(QUANTITY);
```

The code is much cleaner after introducing this constant, but where is a good place for this constant?

Some people tend to put these constants in the same file they are being used in, with the constants at top. For a complex page, the constants will occupy a lot of lines of code to make that class long. The next thing you might do is to move those constants into a `Constants` class; this way the page becomes cleaner. But a constants class does not require the constants to be homogenous--of the same type. We can enforce this using an enum.

Using enums to wrap and organize locators into cohesive groups

You might find your code starts to get quite full of locators—locators everywhere! This technique shows you how to organize them into a single place.

Code is cluttered with long locators that make it hard to understand.

Wrapping locators to their own enum class that acts as a locator factory. Enum was introduced in Java 5; we should take advantage of this language feature and use it to organize our locators to make our code more cohesive. Since enum implicitly extends `Enum`, it cannot extend `By`.

[ShoppingCartBySupplier.java](#)

```

public enum ShoppingCartBySupplier {
    QUANTITY(
        By.name("cartDS.shoppingcart_ROW0_m_orderItemVector_ROW0_m_quantity"));

    private final By by;

    ShoppingCartBySupplier(By by) {
        this.by = by;
    }

    public By get() {
        return by;
    }
}

```

This can then be used as follows:

[ShoppingCartBySupplierIT.java](#)

```
driver.findElement(ShoppingCartBySupplier.QUANTITY.get()) (1)
```

1. Call QUANTITY.get() to retrieve the `By` instance and pass it as the parameter to `findElement`

How cleaner it is when compared to Listing A `By` locator with a really long name.

Real life example

We didn't invent this example, it is from a live website, <http://leanpub.com>. Once you add a book to cart and go to shopping cart page, you can see this quantity input field on that page to allow you to enter a bigger number(or at least we hope so). The shopping cart was built using YUI and it is no longer a choice to do UI. We like Material-UI and we have example in Chapter 18 to automate Material-UI datepicker. When automating datepicker, we use this technique extensively, so it will be helpful if you can master this technique.

A benefit of having an enum is that it can be a great reference for new or inexperienced team members. When they join your project, before adding new constants, they can check whether the constants are already there. If they're not familiar with certain search mechanisms, for example, they can use the existing constants as examples to learn how to produce the ones they want. For example, developers who are interested in learning XPath selectors can check the suppliers to learn how to come up with new locators. By organizing them in one place improves the cohesion of the code and will reduce duplication in the system.

Can you see something we can improve in that example? Let us have a look that enum again. Even the locators are organized properly, it still needs to be call `get` as shown in Listing [ShoppingCartBySupplierIT.java](#). Calling `get` once may not be an issue, but it will become annoying if you need to call it over and over again. Is there a way to just pass the enum itself without calling `get`?

Yes, we will introduce a new technique to simplify this method calling.

Expose locator enums as a `Supplier<By>` interface

With the introduction of the enum constants, we don't need to use the locators with long names any more. It is an improvement. But there is still some problem lingering in the code base we need to address.

As in Listing [ShoppingCartBySupplierIT.java](#), you always need to call the `get` method before you can pass the `By` locator in the enum as the parameter to the finder methods of `Browser` and `Element`. It will add up the complexity of the automation code when many locators are used.

If we can make a call with enum constant `QUANTITY` directly as the parameter for method `findElement`, it will simplify the method calling as,

```
driver.findElement(QUANTITY) (1)
```

The code looks better than the one need to call `get`. And it is not difficult to achieve this.

The solution to this problem is relatively simple. What we need to do is to have the enum implement `Supplier<By>` interface and follow some conversions.

- Organize the same type of `By` locators inside same enum.
- Name the enum `Name`, `Id`, `LinkText` and so on.
- Use string type as constructor parameter and convert the string into the `By` type in the constructor of the enum
- Implement the `get` method from `Supplier<By>` interface and provide a `toString` method

For example, the following is a `ByName` enum with 2 constants defined.

[Name.java](#)

```

public enum Name implements Supplier<By> { (1)
    EMAIL("email"),
    QUANTITY("cartDS.shoppingcart_ROW0_m_orderItemVector_ROW0_m_quantity");

    private final By by;

    ByName(String by) {
        this.by = By.name(by); (2)
    }

    @Override
    public By get() { (3)
        return by;
    }

    @Override
    public String toString() { (4)
        return by.toString();
    }
}

```

1. It implements a `Supplier<By>` interface
2. It constructs a `By.ByName` locator in its constructor
3. It returns the `By.ByName` locator it wraps
4. It provides a `toString()` method for debugging message.

When you use `Name.EMAIL.get()`, it will return you a `By.name("email")` locator and `Name.QUANTITY.get()` will return you a `By.name("cartDS.shoppingcart_ROW0_m_orderItemVector_ROW0_m_quantity")`. But we will change the method parameter to use `Supplier<By>` as parameter so you can pass the enum as parameter without calling its `get` method.

So we change the methods in `Browser` to take `Supplier<By>` as parameter.

DelegatingSearchContext.java

```

@Override
public Element findElement(Supplier<By> by) { (1)
    return new Element( (3)
        super.findElement(by.get()) (2)
    );
}

```

1. Use a `Supplier<By>` as the parameter
2. We still need to call the `get()` method before passing to the `findElement` method of `WebDriver` and `WebElement`
3. We need wrap the `WebElement` using `Element` so this method is available to the caller

Now you can just directly use the enum as parameter.

```
driver.findElement(Name.QUANTITY)
```

You can see, not only we wrap `By` inside an enum, but we created a cleaner interface for other classes to use as well. So you no longer need to call the `get` method of the enum before you can use it as the parameter for finder methods.

When you work on a complex web application, with many things need to be located, the code will be much cleaner without calling `get` method repeatedly. From now on, we will continue to use this technique through out the rest of the book. To save paper, when we list locator enums similar to Listing [Name.java](#), we will omit `get()` and `toString()` methods and just provide the following definition and leave out the rest.

```
EMAIL("email"),
QUANTITY("cartDS.shoppingcart_ROW0_m_orderItemVector_ROW0_m_quantity");
```

You will see extensive usage of this technique in part 3.

Summary

- Using a wrapper can solve a variety of problems.
- The `EventFiringWebDriver` is a built-in wrapped driver, and it's perfect if you want to observe web driver.
- Decorated class can be used in existing code without changing that code. They are more complex, and more powerful. They can be used to change the behavior of web driver.
- You can use a HTTP proxy server to capture HTTP status code. You can also capture other information about the web pages your are testing.

- Providing an interface to access table cells on a `WebElement` representing a HTML table.
- Wrapping `By` locators inside enum and let the enum implement `Supplier<By>` to simplify parameter passing

This is the end of part two. In part three of this book we will look at how you can build a framework for your tests to run in that will reduce the amount of code you need write.

1. https://en.wikipedia.org/wiki/Decorator_pattern
2. http://en.wikipedia.org/wiki/Delegation_pattern
3. <https://bmp.lightbody.net>
4. https://en.wikipedia.org/wiki/Strategy_pattern

Part 3: Page Based Automation Framework

We are going to use the technique we learnt from previous part of the book to write a wrapper framework around WebDriver. This framework is written by experienced enterprise developers based upon lessons learned on the development of many web automation projects. It encapsulates much of the complexity of WebDriver and exposes a clean API to boost productivity and make code easy to understand.

The part of the book will explain the principles behind the evolution of a framework, and use it to show you how you can reduce the complexity of your code by developing an in-house framework.

Chapter 13: Forming a Framework

This chapter covers

- Problems with using `WebDriver` and `WebElement` directly
- Adding more functionality to wrapped `WebDriver` and `WebElement`
- Making framework backwards compatible with existing codebase

Part one and two of this book will have given you enough strategies to deal with most of the common automation techniques that you need for working with web applications. Especially in Chapter 12, we mentioned how to attach more responsibilities to original `WebDriver` by wrapping it inside a `DelegatingWebDriver` as an extension point so you can extend it to have a `BaseUrlDriver` to calculate url based on the base url of the project. In this chapter, we will take it even further and look at how you encapsulate other classes and interfaces and build a productivity framework on top of Selenium WebDriver. We will cover a number of techniques to encapsulate common code into a framework, so that your code is smaller, and more reliable.

We use framework to refer an abstraction layer between automation code written by QA engineers and Selenium WebDriver library. We believe this layer can save you tremendous effort in handling the same concern individually over and over again.

Why would you want to create a framework? Why not use WebDriver directly?

Selenium is a common purpose library used for web automation. It doesn't know how your web application will be designed. So they only provide basic handling for your need. From WebDriver perspective, everything on a web page is an element. An input field is an element, a radio button is an element and a table is an element too. But the behavior and properties of these elements differ dramatically. So we need to use combined actions to handle each element. Also, when working on a medium sized project, there are many test cases for automation engineers to implement. And there are many concerns are very similar or identical. Without a framework, each developer needs to deal with the same situation repeatedly. Even worse, different engineer may come with different solutions for the same kind of problem. It will make the automation code hard to maintain. If you've done a certain amount of coding with WebDriver you probably have come across a number of common problems:

- Code that talks directly to a `WebDriver` can be verbose, especially when dealing with explicit waits.
- Form interaction is standard and can be simplified, but it is not done inside Selenium WebDriver library.
- When automating page flow, the repeated code are noticeable and massive.
- When examine a table with many rows, no guidance on how to make the assertion so it takes unnecessary to fix the changed conditions
- You need to write complex code to deal with elements such as tables or datepickers without a framework.

We will use some example to explain the necessity of introducing a framework and all above mentioned problems will be addressed in this roadmap,

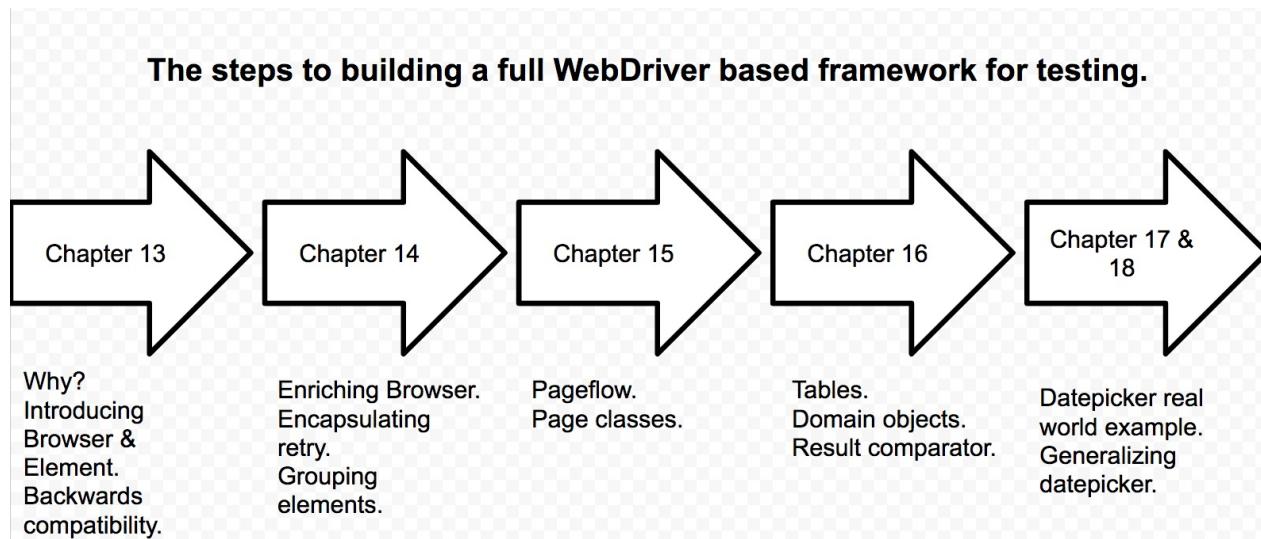


Figure 1. Roadmap of part 3

At the end of part 3 you will have learnt the principles for exposing a well-defined API to simplify your code, with examples of a light-weight framework written to illustrate some principles and practices.

We will use a simple automation task to illustrate the verboseness of dealing with WebDriver directly in your test code. And we will use the techniques illustrated in the following picture to solve the problem gracefully.

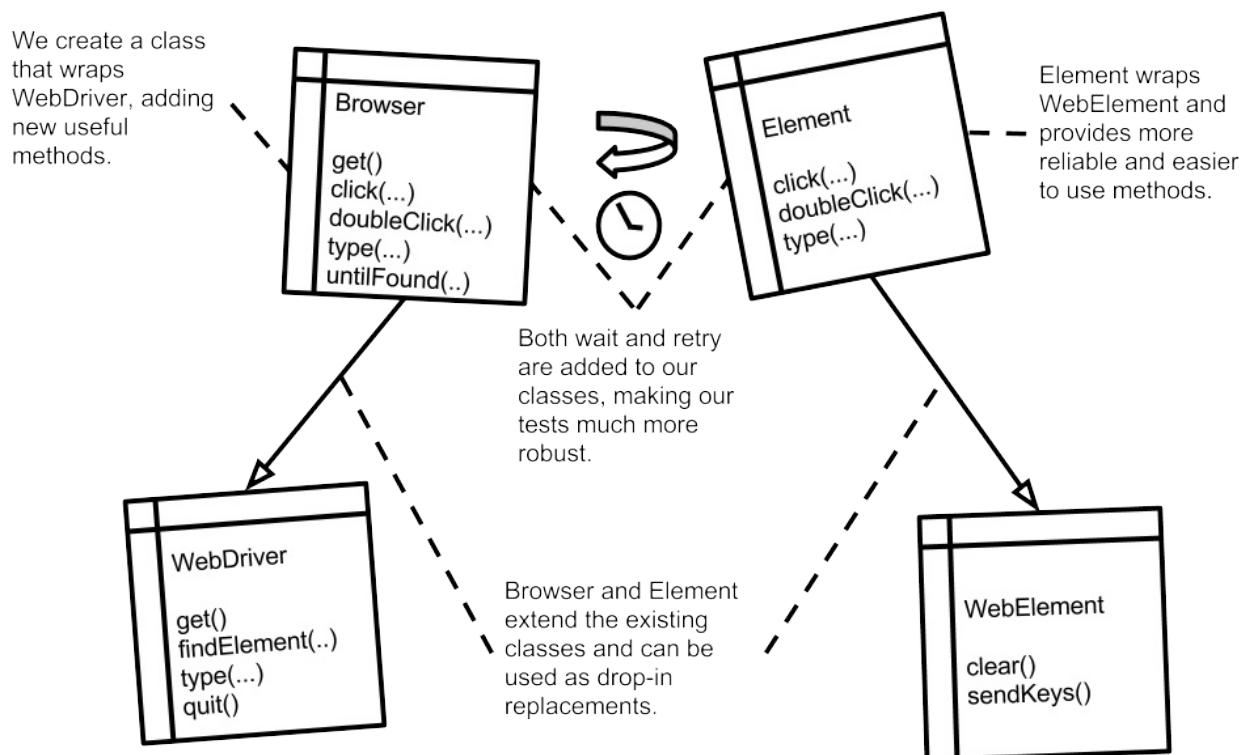


Figure 2. Overview of the framework

At the end of this chapter you will have learnt the principles of how to expose a well-defined API to simplify your code, with examples of a light-weight framework written to illustrate some principles and practices.

Problems with using `WebDriver` and `WebElement` directly

Often, people encounter strange behaviors in automating web applications, as we learnt from Chapter 6. The reason is web technology is very complex and web pages are built upon many layers. Also, modern web pages are usually single page application built using AJAX technology. When automating pages with AJAX technology, additional cautions need to be taken to avoid some common pitfalls. We are going to revisit some of the old tests next.

Locating Elements without using framework code

Consider the location chooser in figure [Location Chooser](#)

<http://localhost:8080/location-chooser.html> below. When you click on the "choose location" button under the "Where is your next vacation?" header, there is a delay as the navigation fades in, if you then choose "MEXICO" the Mexican area fade in over a second or two, finally when you choose a location (such an "Cancun"), there is a further delay as the page reloads.

Where is your next vacation?



Figure 3. Location Chooser <http://localhost:8080/location-chooser.html>

Your first attempt to write an automation code might result in some code like this,

ErrorProneLocatingLogicIT.java

```
driver.findElement(linkText("choose location")).click();
WebElement tabMenu = driver.findElement(By.id("location"));
tabMenu.findElement(linkText("MEXICO")).click();
tabMenu.findElement(linkText("Cancun")).click();           (1)
assertEquals(0, tabMenu.findElements(linkText("Cancun")).size());
assertEquals("Cancun", driver
    .findElement(By.cssSelector(".tools-location strong"))
    .getText());
```

1. The code will fail at this line with an `org.openqa.selenium.NoSuchElementException` but it won't print it out due to the annotation on test

This code looks very normal to most automation engineers. But when you run it, it fails due to a transition effect used to pop up the location choices for user to choose and the code in the listing fails to wait until it fully displays the entire division. When the code is called, the link "Cancun" is still not visible. The `findElement` method in `WebElement` interface throws an `org.openqa.selenium.NoSuchElementException` when the element with the given `By` locator is not found.

```
.NoSuchElementException: Unable to locate element:
{"method":"link text","selector":"Cancun"}
```

You can see the what is the problem, it is a `NoSuchElementException` [1] and it tells you where it is thrown,

By following the stack trace, we can find the code causes the exception and it is this line,

```
tabMenu.findElement(linkText("Cancun")).click();
```

After further analysis, we come to know, this is caused by the timing of the method call is not in sync with the browser. When `WebDriver` fires the `findElement` call, that particular element is indeed not displayed yet. If the `findElement` returns immediately when it doesn't find the element, it will break the automation. Even when the element appears later, the method is already returned with `NoSuchElementException`.

Locating Elements with explicit wait

You may recall that you have learnt from Chapter 6, there is a workaround to this problems from Selenium WebDriver library, to add implicit waiting mechanism to allow `WebDriver` to wait until the element is visible. And we didn't encourage you to use it. We also have learnt from Chapter 6 that we can use explicit waits, as shown in following listing.

LocatingLogicWithExplicitWaitIT.java

```
driver.get("/location-chooser.html");
driver.findElement(linkText("choose location")).click();

WebDriverWait webDriverWait = new WebDriverWait(driver, 5); (1)

WebElement location = webDriverWait.until(
    new Function<WebDriver, WebElement>() { (2)
        @Override
        public WebElement apply(WebDriver driver) {
            return driver.findElement(By.id("location"));
        }
    }
);
```

1. `WebDriverWait` is used to add explicit wait to `WebDriver` and it can be replaced by `FluentWait<WebDriver>`
2. Create an instance of an anonymous inner class of `Function<WebDriver, WebElement>`

When using explicit wait, `WebDriver` will not wait unless you explicitly instruct it to wait. When not specified, as soon as an element can not be found within the current search context, `WebDriver` terminates the search immediately and throws a `NoSuchElementException`. If we predict an element may appear in a late time, due to some transition effect being used to repaint the page, we need to use a `WebDriverWait` or `FluentWait` to specify the maximum wait time for searching for the element before giving up the search and throws a `TimeoutException`, as shown in the following example, we will use both `WebDriverWait` and `FluentWait` in the code, but `FluentWait` is sufficient enough, you can replace `WebDriverWait` with `FluentWait<WebDriver>`.

You need to create a separate `FluentWait<WebElement>` object if the wait is on a `WebElement` object, and each time when you want to wait on something, you need to create an instance of an anonymous inner class,

[LocatingLogicWithExplicitWaitIT.java](#)

```
FluentWait<WebElement> webElementWait          (1)
= new FluentWait<WebElement>(location)
    .withTimeout(1000, MILLISECONDS)           (2)
    .ignoring(NoSuchElementException.class);
WebElement canada = webElementWait.until(
    new Function<WebElement, WebElement>() {      (3)
        @Override
        public WebElement apply(WebElement element) {
            return location.findElement(By.linkText("MEXICO"));
        }
    }
);
canada.click();
WebElement allCanada = webElementWait.until(
    new Function<WebElement, WebElement>() {      (4)
        @Override
        public WebElement apply(WebElement element) {
            return location.findElement(linkText("Cancun"));
        }
    }
);
allCanada.click();
```

1. `FluentWait<WebElement>` to add explicit wait for `WebElement`
2. You will get a `TimeoutException` when you change the value to 10 Milliseconds
3. Create an instance of an anonymous inner class of `Function<WebElement, WebElement>`

4. Create another instance of an anonymous inner class of `Function<WebElement,`
`WebElement>`

You can clearly see from the above example, the disadvantage of explicit wait is too verbose. The code increases in size after you add explicit waiting. What might help is a web driver that provides a method to encapsulate this waiting logic. That's one of the motivations of creating a framework - to provide this kind of solution behind the scene. We are going to show you some techniques used in the development of a framework.

Handling Transition Effect

If a page uses CSS Transition [2], it will likely cause some intermittent failure for `findElement` call. The only way to handle this situation is to use waiting mechanism to delay the search. We learnt in Chapter 6 that we can use explicit wait to retry the search behind the scene. Now we are going to show you is to wrap the explicit wait inside a method so it becomes easier for you to use search with wait. We will observe another test case failure caused by transition effect in Chapter 18, when we automating Material-UI datepicker. Due to the maturity of the framework. The solution shown in Chapter 18 will be very concise.

Adding more functionality to wrapped `WebDriver` and `WebElement`

In Chapter 12, we learnt that by wrapping `WebDriver` and `WebElement`, we can attach additional responsibility to these interfaces and make them easier to use. By applying the same design pattern, we can hide the code which is used to handle explicit wait and remove those cumbersome logic. We already started building the framework in Chapter 12, and right now, it becomes clearer that we need to continue evolving the framework to provide more supports for the automation developers. We can add support in wrapped `WebDriver` and `WebElement` to add one method with the build-in explicit wait, so developers can just call that method if they want the automation code to wait for an element to appear on the web page.

Encapsulating `FluentWait` class to provide built in waiting

We will use this technique to simplify the code in Listing [LocatingLogicWithExplicitWaitIT.java](#) and Listing [LocatingLogicWithExplicitWaitIT.java](#), this technique works also for people not using Java 8, just the code will be some duplicates since **default method** is not available until Java 8, which helps in reducing some duplicates.

You have duplicated code related to waiting. In Listing [LocatingLogicWithExplicitWaitIT.java](#) and Listing [LocatingLogicWithExplicitWaitIT.java](#), the following block has been repeated 3 times with only small variant, such as `By.id("location")` in the code snippet below and `By.linkText("MEXICO")` in other place.

```
WebElement location = webDriverWait.until(
    new Function<WebDriver, WebElement>() {
        @Override
        public WebElement apply(WebDriver driver) {
            return driver.findElement(By.id("location"));
        }
    });

```

This repeated code block with only small difference in each block makes the code unpleasant to read and fragile to maintain.

Wrap both `WebElement` and `WebDriver` in new objects, and provide new methods to support waiting. Wrapping an object to simplify it is known as the **Facade pattern** [3]. You can create an object that has the methods that you want, and delegates calls to the "delegate".

Well, how does this help us with waiting? We'd like to have a new method that allows us to find an element, but wait if it is not there. We've already got a method that supports finding elements, so you need to add another that supports waiting. You need to add the same code to two classes. How can you do this? Java 8 allows interfaces to have **default methods** [4]. An interface with a default method is very similar to an abstract class with a concrete methods while a class can only extend one abstract class but implements multiple interfaces.

What we do in Java 8

Due to the limitation with abstract class, we decided to use an interface for the purpose of adding explicit wait, but this works only in Java 8.

[ExplicitWait.java](#)

```

public interface ExplicitWait {    (4)

    WebElement findElement(Supplier<By> by); (1)

    default (2)
    WebElement await(Supplier<By> by) { (5)
        return new FluentWait<>(this) (6)
            .withTimeout(5, SECONDS)
            .pollingEvery(10, MILLISECONDS)
            .ignoring(NoSuchElementException.class)
            .until((ExplicitWait e) -> findElement(by)); (3)
    }
}

```

1. Classes that implement this interface must implement this method.
2. The `default` keyword means that every implementing class will have this method.
We name this method `await` but you can name it `findElementWithTimeout` or anything makes sense to you and your team.
3. You provide an additional method to find methods using wait.
4. The class implements this interface can still extend other abstract class
5. We use `Supplier<By>` as parameter so it take the enums as parameter.
6. The diamond operator is for type inference operation

Type inference

Type inference refers to the implicitly deduction of the class type of an expression in a programming language. The Diamond Operator `<>` reduces some of Java's verbosity surrounding generics by having the compiler infer parameter types for constructors of generic classes.

In the following snippet, `this` is an `ExplicitWait`,

```
new FluentWait<>(this)
```

So it can be inferred that the what inside the `<>` operation is an `ExplicitWait`

```
new FluentWait<ExplicitWait>(this)
```

This works only after Java 7.

Why we call the method in `ExplicitWait await`

We introduced a technique in Chapter 2 to use chained locators, so we can chain the search methods. If we called this method `findElementWithWait`, it would be too long when we chained multiple method together. For example, this code snippet uses `findElementWithWait`, we need to break it into two lines, otherwise it will run over.

```
browser.findElementWithWait(CALENDAR)
    .findElementWithWait(PREV_MONTH_BUTTON)
```

While the one using `await` can have two chained methods on one line,

```
browser.await(CALENDAR).await(PREV_MONTH_BUTTON)
```

You can see `await` is shorter and cleaner. Also `await` is meaningful to identify a wait. `FluentWait` just has two `until` methods, one takes a `Predicate` as parameter and another takes a `Function`. But in `ExplicitWait` interface, we use `await` to make it distinguishable from the `until` methods in `FluentWait`.

Both `Browser` and `Element` already have the `findElement` method. To get the new `await` method, they can implement `ExplicitWait` interface. Now let us look at these two classes, after the change, they both have the `await` method now.

`Browser` class implements `ExplicitWait` so it has `await` method,

Browser.java

```
public class Browser extends DelegatingWebDriver
    implements ExplicitWait {    (2)

    public Browser(WebDriver driver) {    (1)
        super(driver);
    }
}
```

1. You wrap up an existing driver.

2. Since `Browser` implements `ExplicitWait`, it inherently has the `await` method

`Element` class implements `ExplicitWait` so it also inherits the `await` methods,

Element.java

```

public class Element implements ExplicitWait { (2)

    private final WebElement webElement;

    public Element(WebElement webElement) { (1)
        this.webElement = webElement;
    }

    @Override
    public WebElement findElement(Supplier<By> by) {
        return webElement.findElement(by);
    }
}

```

1. Wrap the existing WebElement.
2. It implements `ExplicitWait` so it has `await` method

Now we can call the `await` method from both `Browser` and `Element`.

We collect the locators into `LinkText` enum.

```

CHOOSE_LOCATION("choose location"),
MEXICO("MEXICO"),
CANCUN("Cancun"),

```

And add `LOCATION` to `Id` locator supplier enum,

```

LOCATION("location"),

```

You can see in the following test, `await` can be called from both `Browser` and `Element`, and this test is much smaller than Listing [LocatingLogicWithExplicitWaitIT.java](#), only one quarter of the code, which explains why we want to use framework.

[ExplicitWait_v1_IT.java](#)

```

browser.get("/location-chooser.html");
browser.await(CHOOSE_LOCATION).click(); (1)
Element tabMenu = new Element(browser.await(LOCATION)); (2)
tabMenu.await(MEXICO).click(); (3)
tabMenu.await(CANCUN).click();
assertEquals("Cancun", browser.await(TOOLS_LOCATION_STRONG)
    .getText());

```

1. await is available from browser
2. await is available from element
3. But since the `await` returns `WebElement` so we need to create a `Element`

Even if you still use pre-Java 8 version, you can still use this technique by using Google Guava library. You can download it by adding the following section to `pom.xml` file,

Dependency for Google Guava library

```
<dependency> (1)
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>19.0</version>
</dependency>
```

1. Added this block anywhere inside the `<dependencies>...</dependencies>` block

Guava Library

Guava library was a nice library to provide some nice Java 8 features. Java 8 has similar classes and interfaces in `java.util.function` package. You don't need to add the dependency block when you use Selenium WebDriver. WebDriver has a dependency on Guava library so the project using Selenium automatically has the dependency.

And alternatively, how can we implement it in project using pre-Java 8 version.

What we do in pre-Java 8

If you don't use Java 8, you need to define an empty `await` method in `ExplicitWait` interface and provide the implementation in each class which implements `ExplicitWait`,

[ExplicitWait.java](#)

```
public interface ExplicitWait {
    WebElement findElement(Supplier<By> by);
    WebElement await(Supplier<By> by); (1)
}
```

1. An empty method

Then we implement `public WebElement await(Supplier<By> by)` in both `Browser` and `Element` class,

[Browser.java](#)

```

public WebElement await(final Supplier<By> by) { (1)
    return new FluentWait<>((ExplicitWait) this) (3)
        .withTimeout(10, TimeUnit.SECONDS)
        .pollingEvery(100, TimeUnit.MILLISECONDS)
        .ignoring(NoSuchElementException.class)
        .until(new Function<ExplicitWait, WebElement>() { (4)
            @Override
            public WebElement apply(ExplicitWait browser) {
                return browser.findElement(by);
            }
        }); (2)
}

```

1. Implement the method from `ExplicitWait`
2. Need to create anonymous inner class of `Function<Browser, WebElement>` if you don't use Java 8
3. This is called type inference using Diamond Operator and the result is a `Browser`
4. Since the `FluentWait` is with `Browser` type, so the parameter for `until` needs to be a `Function<Browser, WebElement>` type

You can see another advantage of using Java 8 is not to repeat the implementation if they can use the default one in the interface, even they are not exactly same. You can still introduce a super class to remove this duplicated methods.

Regardless whether you use Java 8 or not, the test is same as Listing [ExplicitWait_v1_IT.java](#).

The examples give us a clear idea of some of the benefits of wrapping up common WebDriver classes. But you need to invoke the constructor of the `Element` every time you use the `await` method.

```
Element tabMenu = new Element(browser.await(Id.LOCATION));
```

The reason is when `await` method returns a `WebElement` interface, which is an original interface from WebDriver library, you lose the wrapping you have done around it. To overcome this shortcoming, `await` method need to return an `Element` instead, that way, you can continue using the wrapped `WebElement` and you can take advantage of

the functionalities attached to `Element`. It means the framework code in Listing [less-optimal] is not optimal, we need to improve it to have it returning an `Element` rather than a `WebElement`.

So we are making the change, the new code is shown in the following listing,

`WebElement` is replaced by `Element`,

[ExplicitWait.java](#)

```
public interface ExplicitWait {  
  
    Element findElement(Supplier<By> by); (1)  
  
    default Element await(Supplier<By> by) {...} (2)  
}
```

1. Return `Element` instead of `WebElement` now
2. Return `Element` instead of `WebElement` now

So we can assign the result of `await` to an `Element` without calling `new Element(...)` and it saves a lot of boiler plate code.

```
Element tabMenu = browser.await(Id.LOCATION);
```

Here is the sequence diagram of the `await` method.

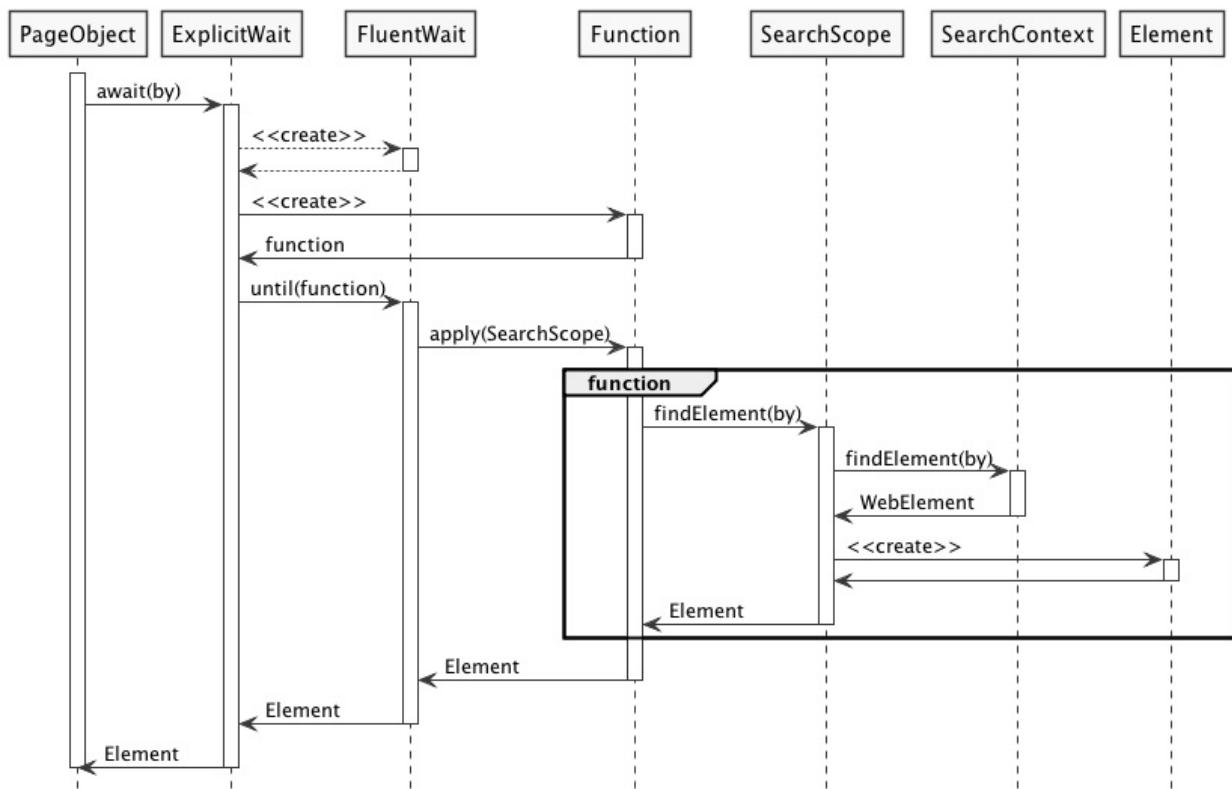


Figure 4. Sequence diagram `await` method

We can attach more responsibility to this `Browser` and `Element` classes to make them more versatile.

Using an `Optional` class To Check For Element Presence

There is situation that some element may appear or may not. To handle this situation gracefully, we are going to provide another method to use an `Optional` class as an wrapper for the element we are trying to find as the return type. When nothing found, the `isPresent()` method of that `Optional` object returns `false`, when it is found, its `get()` method will return the element. This technique works better with Java 8 installation, since it is a built interface from the JDK.

Then you can use the `Optional` class from Guava library. It provides same functionality as the one from Java 8. Here is the JavaDoc for Guava `optional` class [5]

Code for checking for the presence of an element must catch `NoSuchElementException`. This adds several lines of code to each use.

Provide an `optionalElement` method to find optional elements. Again, you want to create a method that you could add to both `Browser` and to `Element`. This can be done as per listing `SearchScope.java` below. If you haven't use Java 8 yet, the default method is not available in earlier version of Java. You need to have an empty method and repeat the implementation in each of the class which implements this interface

[SearchScope.java](#)

```
public interface SearchScope {

    Element findElement(Supplier<By> by);

    default Optional<Element> optionalElement(Supplier<By> by) { (1)
        try {
            return Optional.of(findElement(by));
        } catch (NoSuchElementException ignored) {
            return Optional.empty(); (2)
        }
    }
}
```

1. Only available in Java 8 or via Guava library
2. This is from `java.util.Optional`, it is called `absent()` in Guava `Optional`

You can then update `Element` and `Browser` to implement it, in the same manner you did for `ExplicitWait` previously. Since there is a `findElement` method in `SearchScope` interface, we can have `ExplicitWait` extending `SearchScope` and remove the `findElement` method from `ExplicitWait` interface.

But in older version of Java, `SearchScope` is like this,

[SearchScope.java](#)

```
import org.openqa.selenium.By;
import com.google.common.base.Optional; (2)

public interface SearchScope {
    Element findElement(Supplier<By> by);
    Optional<Element> optionalElement(Supplier<By> by); (1)
}
```

1. Pre-Java 8 interface
2. Import an interface from Google Guava library

And the classes implementing `SearchScope` interface would have to repeat the same implementation, just as illustrated by the following code snippets,

Pre-Java 8 implementation

```

public class Element implements SearchScope {
    ...
    public Optional<Element> optionalElement(Supplier<By> by) { (1)
        try {
            return Optional.of(findElement(by));
        } catch (NoSuchElementException ignored) {
            return Optional.absent(); (2)
        }
    }
}

```

1. `Browser` class has exactly the same implementation
2. The `empty()` method of `java.util.Optional` is call `absent()` in Guava `Optional` class.

The same `public Optional<Element> optionalElement(By by)` is repeated in `Browser`,

Pre-Java 8 implementation

```

public class Browser implements SearchScope {
    ...
    public Optional<Element> optionalElement(Supplier<By> by) {
        ... (1)
    }
}

```

1. Exactly same as listing Pre-Java 8 implementation

Then you can call the `isPresent` method to check whether you has found something using `Supplier<By>`. If `isPresent` is true, it means you have found an element, otherwise not.

Also you can see `click` method is used multiple times in Listing [ExplicitWait_v1_IT.java](#), so you can add these method to `ExplicitWait` for the time being. We are going to show another technique next.

Adding helper methods for some frequently used method call.

In Listing [ExplicitWait_v1_IT.java](#), we can see there are three occurrences of `await` method call followed by calling `click`. These two methods are frequently used together in web automation code. Also you need to call `getText` after calling `await`.

Often, we need to find an element and click it using `await` method call followed by calling `click`. Also we need to find an element and read its text using `getText` after calling `untilFound`. And we also check whether an element can be found by using a `By` locator. These chained methods calls are very frequent and they are repeated everywhere.

We can achieve this by adding helper methods to give a single method call so you no longer expose a chained method calls in the code,

```
default void click(Supplier<By> by) {
    await(by).click();
}

default String getText(Supplier<By> by) {
    return await(by).getText();
}
```

And add this method into `SearchScope`, it can be used to check whether an element can be found using the giving locator enum, same as by calling `optionalElement(by).isPresent()` in a method chain.

```
default boolean isPresent(Supplier<By> by) {
    return optionalElement(by).isPresent();
}
```

Then we can replace the chained method calls in Listing [ExplicitWait_v1_IT.java](#) and add another line to check whether the link for `Cancun` is still there. The test asserts it shouldn't be there, since it should close after being clicked.

[ExplicitWait_v2_TT.java](#)

```
browser.click(CHOSE_LOCATION);
Element tabMenu = browser.await(LOCATION);
tabMenu.click(MEXICO);
tabMenu.click(CANCUN);
assertEquals("Cancun", browser.getText(TOOLS_LOCATION_STRONG));
assertFalse(tabMenu.isPresent(CANCUN));           (1)
```

1. Here you can check that "Cancun" is not present.

You can see the `isPresent`, `click` and `getText` methods makes the code cleaner. This block of code provides the same functionality of Listing [LocatingLogicWithExplicitWaitIT.java](#) and Listing [LocatingLogicWithExplicitWaitIT.java](#)

together.

```
Element.getAttribute("value") is also used frequently, we can add following method to  
DelegatingWebElement ,
```

```
public String getValue() {  
    return delegate.getAttribute("value");  
}
```

Then we can just call `Element.getValue()`.

The reason we add these helper methods into `ExplicitWait` class is because both `Browser` and `Element` implements `ExplicitWait` interface, the helper methods in `ExplicitWait` are available for them. And we make code simpler and pleasant to read.

There are other examples of using framework to add functionality. By introducing a `Browser` class, you can attach other useful methods to it simplifies the calling code, as illustrated by the following table:

Table 1. Compare code with and without using framework

	Without framework	With framework	Benefit
Clicking OK on a message alert	<code>driver.switchTo().alert().accept();</code>	<code>browser.acceptAlert();</code>	Hide the method chain
Drag and Drop	<code>new Actions(driver).dragAndDrop(findElement(from), findElement(to)).perform();</code>	<code>browser.dragAndDrop(from, to);</code>	More concise code

So instead of using following chained method calls,

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

You can just use one method invocation, which is much shorter and concise by hiding the method chain.

```
browser.acceptAlert();
```

Also, you don't need to have this lengthy method with object creation and method invocation,

```
new Actions(driver).dragAndDrop(findElement(from), findElement(to)).perform();
```

You can just use a single method to achieve the same functionality,

```
browser.dragAndDrop(from, to);
```

That's the beauty of using frameworks.

Making framework backwards compatible with existing codebase

Before we develop this framework, there may already exist some tests with the method taking `WebDriver` and `WebElement` interfaces as parameter, we would like to use the `Browser` and `Element` interfaces as the parameter to those code, so the investment we made in the past can be recycled to take advantage of this new framework. You may wonder why we need to do this, let us take a look of the following test.

```
driver.switchTo().frame(editorFrame);
```

We are trying to switch to the iframe but it will fail with `IllegalArgumentException`. When you run the test, it prints the following stack trace,

```
java.lang.IllegalArgumentException: Argument is of an illegal type:  
swb.ch13framework.v2.Element  
    at WebElementToJsonConverter.apply(WebElementToJsonConverter.java:81)  
    at RemoteTargetLocator.frame(RemoteWebDriver.java:1003)  
    at WysiwygInputFailureIT.failToSwitchToIframe(WysiwygInputFailureIT.java:35)
```

We follow the stack trace to WebDriver source code and find the exception is thrown by the `apply` method of `WebElementToJsonConverter` class.

[WebElementToJsonConverter.java](#)

```

while (arg instanceof WrapsElement) {      (1)
    arg = ((WrapsElement) arg).getWrappedElement();  (3)
}
...
throw new IllegalArgumentException("Argument is of an illegal type: " + (2)
    arg.getClass().getName());

```

1. This is to extract the inner most `WrapsElement` if it is wrapped many times.
2. The exception is thrown here
3. From this logic we can know, as long as we have `DelegatingWebDriver` implementing `WrapsElement`, it will use what is wrapped inside.

From its logic, we know it expects that our class that wraps `WebElement` needs to implement `WrapsElement` interface. We are going to do this to `WebDriver` and `WebElement` interfaces to make them backwards compatible.

Making Browser Backwards Compatible By Implementing WebDriver

First let us make `Browser` class backwards compatible so it can be used in the code where expecting a `WebDriver` interface as parameter.

You have a `Browser` class which can use for new tests, and you'd like to use it in existing tests, so you only have to manage one driver.

To do this you can add interface `WrapsDriver` to `DelegatingWebDriver` class as per listing [DelegatingWebDriver.java](#) below. You can then sub-class it to override specific methods.

[DelegatingWebDriver.java](#)

```
public class DelegatingWebDriver
    implements WebDriver, JavascriptExecutor, TakesScreenshot,
    HasInputDevices, HasCapabilities, WrapsDriver { (4)
    private final WebDriver delegate; (1)

    public DelegatingWebDriver(WebDriver delegate) {
        this.delegate = delegate;
    }
    ... (2)
    @Override
    public WebDriver getWrappedDriver() { (3)
        return ((WrapsDriver)driver).getWrappedDriver();
    }
}
```

1. Store the delegate.
2. The remaining methods all delegate their calls to `WebDriver delegate`.
3. This method is from interface `WrapsDriver`
4. Add `WrapsDriver` interface as one of the interfaces

After this change, this class is safe to be used for framework code and we made it backwards compatible with existing framework. With this class, you can sub-class it, and only override the methods you need to change the behavior of.

Browser.java

```
public class Browser extends DelegatingWebDriver {

    public Browser(WebDriver driver) {
        super(driver);
    } (1)
}
```

1. The constructor of the `Browser` class

By applying Object Oriented principle, we made `Browser` backwards compatible with old tests. Let us go back to chapter 5, Making maintainable tests using the Page Object pattern, and take another look of Listing 5.8 [d/SearchPage.java](#). In the listing, `SearchPage` class takes a `WebDriver` interface as the constructor parameter, now we are going to pass it a `Browser` object. Will it work? Let us see.

We create this new test `SearchPageV2IT` and run it with a `BrowserRunner` runner class so it will inject a `Browser` into the test, we then create a `SearchPage` using the `browser` variable,

SearchPage_v2_IT.java

```
import swb.ch05pageobjects.d.SearchPage; (1)

@RunWith(BrowserRunner.class)
public class SearchPage_v2_IT {
    @Inject
    private Browser browser;

    @Test
    public void search() throws Exception {
        SearchPage searchForm = new SearchPage(browser); (2)
        searchForm.searchFor("funny cats");
    }
}
```

1. This `swb.ch05pageobjects.d.SearchPage` class is for old tests and it expects `WebDriver` interface as the constructor parameter
2. We pass a `Browser` class as parameter now and it still works

The test passes, it works! By making `Browser` implementing `WebDriver`, we make it backwards compatible with many existing Page Object we had written in the past.

Now let us make `Element` class backwards compatible so it can be used in the code where expecting a `WebElement` interface as parameter.

Making Element Backwards Compatible By Implementing WrapsElement interface

It is the same case with `Element` class too.

You have an `Element` class which can use for new tests, and you'd like to use it in existing tests, so you only have to manage one element.

Again, use the delegate pattern to delegate the code to another class. To do this you can create a new class `DelegatingWebElement` as per listing [DelegatingWebElement.java](#) below and have it implementing `WrapsElement` interface. You can then sub-class it to override specific methods.

[DelegatingWebElement.java](#)

```
public class DelegatingWebElement implements WebElement, WrapsElement {    (3)
    private final WebElement delegate;

    public DelegatingWebElement(WebElement delegate) {
        this.delegate = delegate;    (1)
    }
    ... (2)
    @Override
    public WebElement getWrappedElement() {
        return delegate;            (4)
    }
}
```

1. Store the delegate.
2. The remaining methods all delegate their calls.
3. Add `WrapsElement` as one of the interfaces
4. Return the wrapped `WebElement`

With this class, you can sub-class it, and only override the methods you need to change the behavior of.

[Element.java](#)

```
public class Element extends DelegatingWebElement {

    public Element(WebElement delegate) {    (1)
        super(delegate);
    }
}
```

1. The constructor of the `Element` class

Creating a delegate allows us to override specific methods, and yet keep our code readable. In this example you have use it to extend `Element` to allow it to be used as a drop-in and backwards compatible replacement for `WebElement` in existing code.

Let is use `LoginForm` class in Listing 5.2 [LoginForm.java](#) from Chapter 5, and create new test using `Browser` and `Element` and this old `LoginForm`,

[LoginForm_v2_IT.java](#)

```

@Test
public void checkLoginForm() throws Exception {
    Element login = driver.await(LOGIN); (2)

    LoginForm loginForm = new LoginForm(login); (1)
    loginForm.loginAs("foo@bar.com", "secret");

    new WebDriverWait(driver, 1) (3)
        .until(ExpectedConditions.titleIs("You Are Logged In"));
}

```

1. It uses the `swb.ch05pageobjects.LoginForm` class created in Chapter 5 that takes `WebElement` as constructor parameter
2. Now we pass an `Element` object to its constructor
3. `WebDriverWait` expects `WebDriver` but we pass in `Browser`

And it still works so we achieved our goal of making them compatible with the old tests.

Introduce base class to reduce duplicated code

Let us look at these two methods,

Methods in both `DelegatingWebDriver.java` and `DelegatingWebElement.java`

```

@Override
public List<WebElement> findElements(Supplier<By> by) {
    return delegate.findElements(by);
}

@Override
public Element findElement(Supplier<By> by) { (4)
    return new Element(delegate.findElement(by));
}

```

You can see from Listing `DelegatingWebDriver.java` and `DelegatingWebElement.java`, they belong to both `DelegatingWebDriver` and `DelegatingWebElement`.

And since these two methods are from interface `SearchContext`, we can implement that interface to introduce `DelegatingSearchContext` as super class for `DelegatingWebDriver` and `DelegatingWebElement` and take advantage of Java Generics to have this class take a type, which can be any subtype of `SearchContext`, either a `WebDriver` or a `WebElement`,

so in the subclass of `DelegatingWebDriver`, where we use `WebDriver` as the generic type parameter, `T` in `protect final T delegate` is a `WebDriver` class. But in `DelegatingWebElement`, it is `WebElement`. If you just declared it as `SearchContext`, you would need to cast it to `WebDriver` in `DelegatingWebElement` and `WebElement` in `DelegatingWebElement` before you could call the methods of each interface.

[DelegatingSearchContext.java](#)

```
public class DelegatingSearchContext<T extends SearchContext> (1)
    implements SearchContext, ExplicitWait {
    protected final T delegate; (2)

    public DelegatingSearchContext(T delegate) {
        this.delegate = delegate;
    }
    ... (3)
}
```

1. The type parameter of this class is any subtype of `SearchContext`
2. Make this variable protected so it is accessible from its subclass
3. We implement `findElements`, `findElement` methods and remove them from both `DelegatingWebDriver` and `DelegatingWebElement`

Then we can have `DelegatingWebDriver` extending `DelegatingSearchContext` with type parameter `WebDriver`, delete the variable `WebDriver driver` from it, and delete those two methods in Listing [Methods in both DelegatingWebDriver.java and DelegatingWebElement.java](#) from `DelegatingWebDriver`.

[DelegatingWebDriver.java](#)

```
public class DelegatingWebDriver
    extends DelegatingSearchContext<WebDriver> (1)
    implements WebDriver, JavascriptExecutor, TakesScreenshot,
    HasInputDevices, HasCapabilities, WrapsDriver {

    public DelegatingWebDriver(WebDriver delegate) {
        super(delegate); (2)
    }
    ... (3)
}
```

1. The type for `DelegatingWebDriver` is `WebDriver`
2. Call constructor of super class and pass the instance of `WebDriver`

3. The rest of the methods also remain unchanged

Here is the class diagram of `Browser` class and its ancestors from WebDriver.

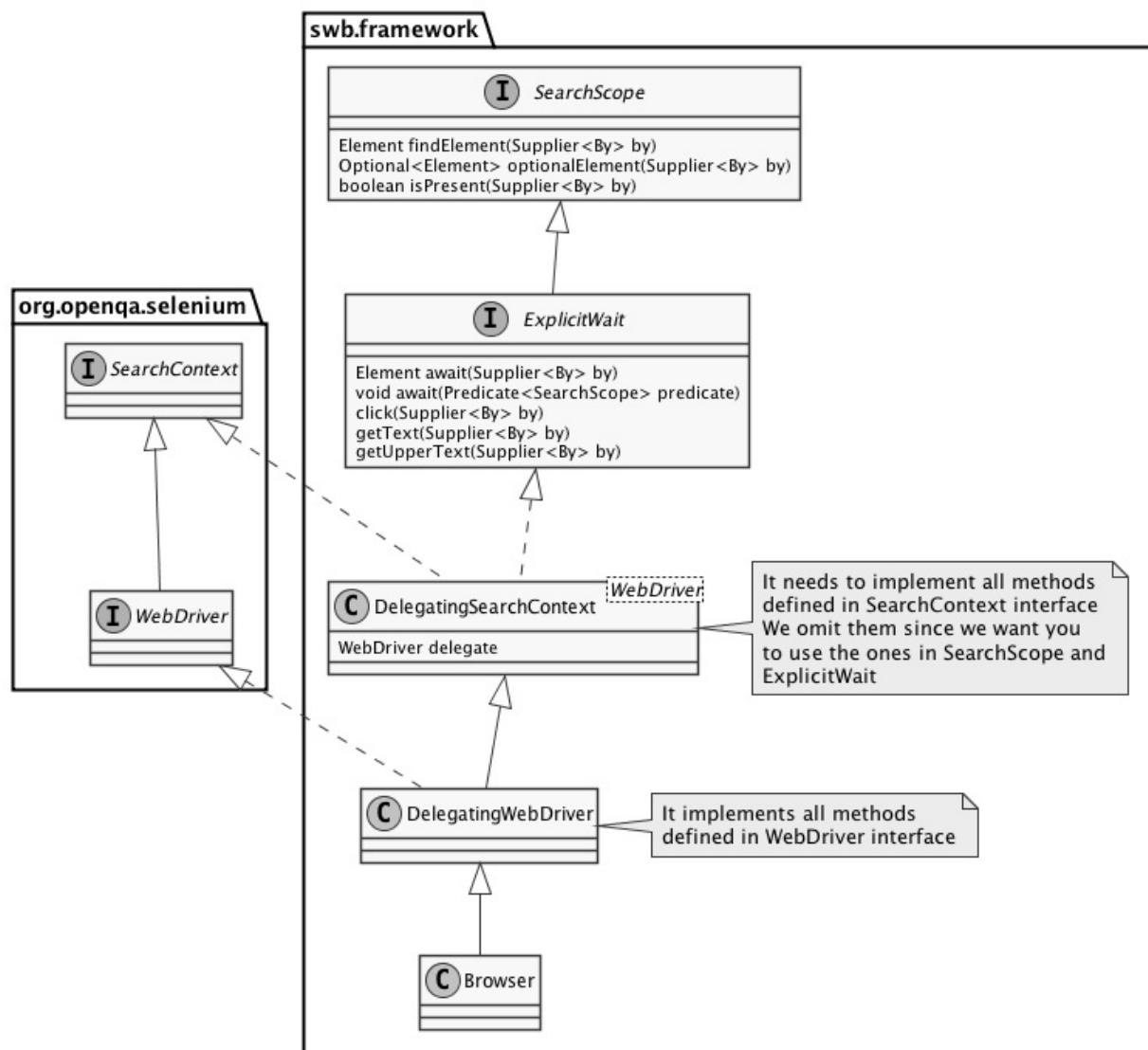


Figure 5. Class diagram of the Browser and its ancestors

And have `DelegatingWebElement` extending `DelegatingSearchContext` with type parameter `WebElement`, delete the variable `WebElement element` from it, and delete those two methods in Listing [Methods in both DelegatingWebDriver.java and DelegatingWebElement.java](#) from `DelegatingWebElement`.

DelegatingWebElement.java

```

public class DelegatingWebElement
    extends DelegatingSearchContext<WebElement> (1)
    implements WebElement {

    public DelegatingWebElement(WebElement delegate) {
        super(delegate); (2)
    }
    ... (3)
}

```

1. The type for DelegatingWebElement is WebElement
2. Call constructor of super class and pass the instance of WebElement
3. The rest of the methods also remain unchanged

And here is the class diagram of Element class and its ancestors from WebDriver.

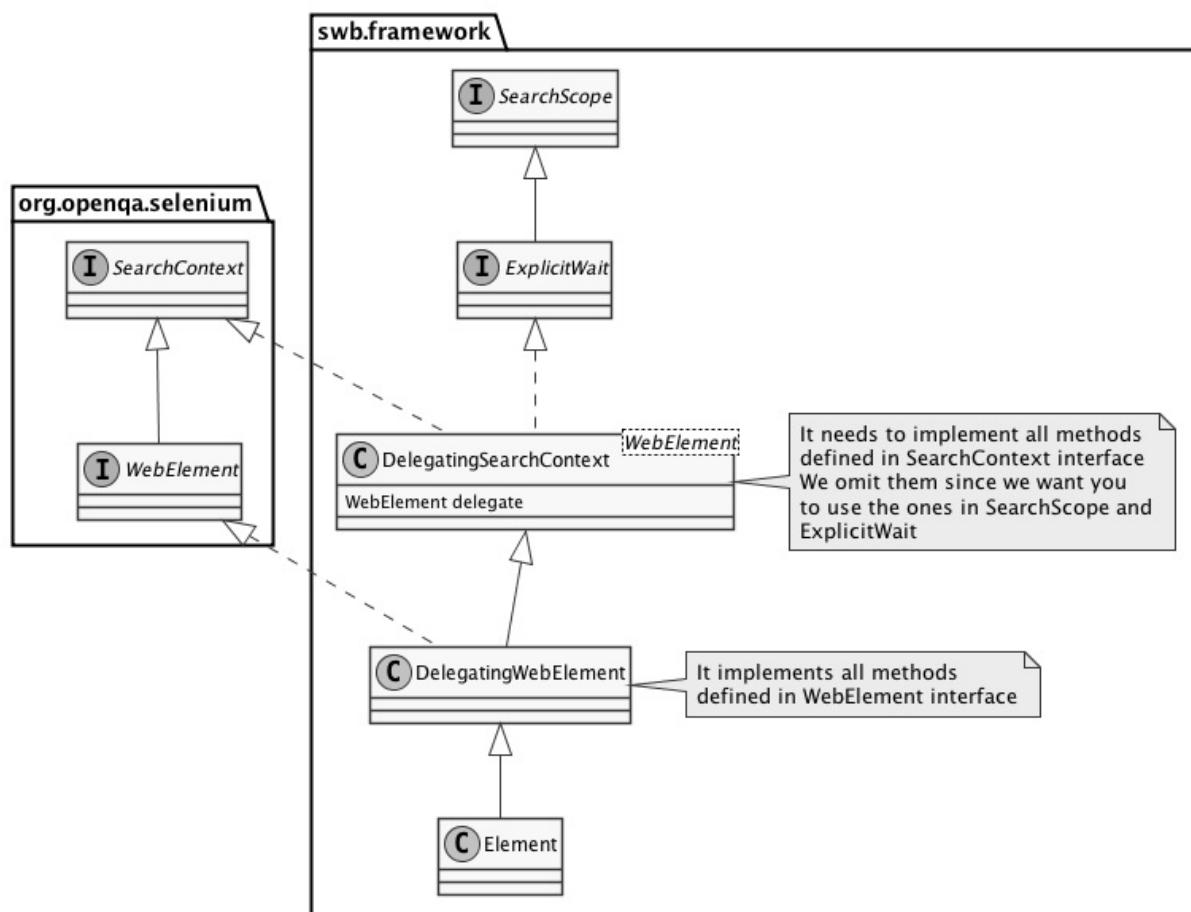


Figure 6. Class diagram of the Element and its ancestors

Now the framework evolved to a better state, comparing to the beginning of this chapter. It will continue evolve in the coming chapters.

Summary

- Code that talks directly to `WebDriver` can be reduced in size
- Wrapping `WebDriver` allows you alter the behavior of existing method
- Wrapping allows you extend `WebDriver` with new methods
- You can make these change, while at the same time keeping the code backwards compatible

In the next chapter you will learn how to encapsulate some common operations related to form elements and how to organize the elements into logical groups to simplify the page object.

1. http://seleniumhq.org/exceptions/no_such_element.html
2. <https://www.w3.org/TR/css3-transitions/>
3. http://en.wikipedia.org/wiki/Facade_pattern
4. <https://docs.oracle.com/javase/tutorial/java/landI/defaultmethods.html>
5. <http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/base/Optional.html>

Chapter 14: Encapsulating and Grouping elements

This chapter covers

- Enriching `Browser` class to simplify the interaction of form elements
- Encapsulating a retry mechanism to make tests robust
- Organizing elements into logical groups

In the last chapter we learnt that by encapsulating explicit wait into `ExplicitWait` interface, and having the `DelegateSearchContext` classes to implement it, so both `Browser` and `Element` classes also inherently implement it, we reduced the amount of code you need to complete some tasks.

In this chapter you will see how to add more classes and methods which will enrich your framework and help you to support many of the common types of element you'll encounter when automating and testing your applications. For example, we are going to add method to allow you to set a value to an input field with only one method call, rather than going through a series steps: finding that element, clearing its text and sending keys. We will also show you how to add other methods to handle radio buttons, checkboxes and selects. This will make it easy and quicker to write tests and your tests will be more robust.

By the end of the chapter you will have learned how to encapsulate common form elements, how to create a class that will automatically retry on failure, and how to break a page object down into simpler classes.

Enriching `Browser` class to simplify the interaction of form elements

Part of the automation is dealing with various form elements, for example, `input`, `textarea`, `checkbox`, `select`, `radio` and `button` (or `input type=button`), as in the Registration Form from Chapter 3. It has all kinds of fields we need for a from,

Registration Form

Email

E.g. john.doe@swip.com

text input

password

Password

Confirm Password

How did you hear about us?

-

select

Channel

Contact me by email. Contact me by phone.

radio

Frequency

Contact me weekly. Contact me daily. Contact me hourly.

Please choose what you are interested in

Books
Music
Movies

multiple select

Tell us more about yourself

checkbox

textarea

I accept the terms and conditions

Sign-up

button

Figure 1. Registration Form

In Chapter 3, Interacting with form elements on a page, we have shown you how to use the `sendKeys` and other methods to type in texts into an input field. There isn't a single way to set a form input's value. You always need these 3 steps,

1. Locate it as an instance of `Element` first
2. Clear its contents and
3. Then use the `sendKeys` method to type in the value.

And we can review what we learnt from last chapter in the following figure, to see what methods we can use to improve the code from Chapter 3,

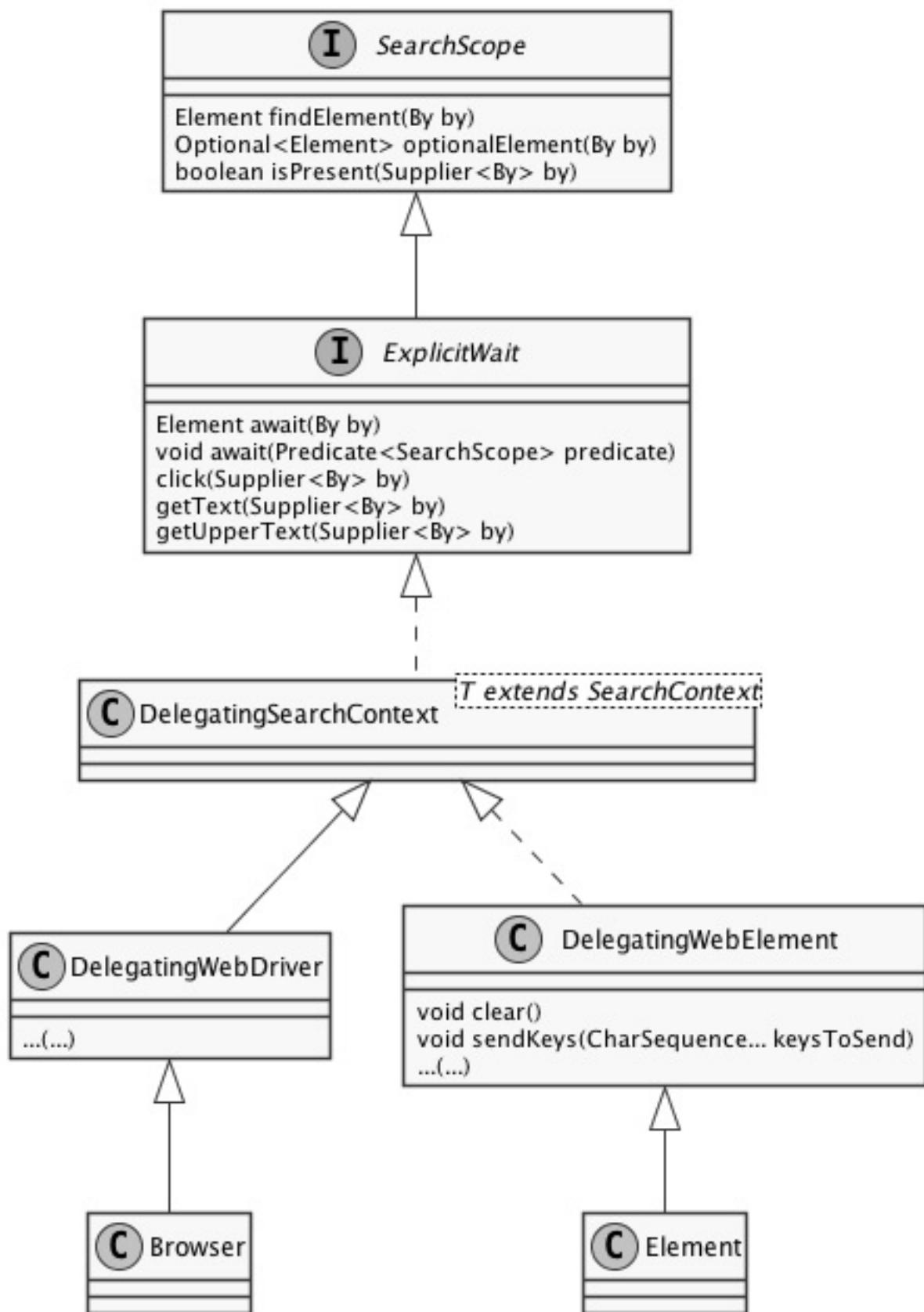
Figure 2. Simplified class diagram of `Browser` and `Element`

Table 1. Steps used to enter text into an input field

Step	Method call	Defined in
Locating "email" Element	browser.await(By.name("email"))	ExplicitWait
Clearing old contents	email.clear()	DelegatingWebElement
Typing in new value	email.sendKeys("john@doe.com")	DelegatingWebElement

And we can use the following code snippet to set a user name and password on the registration form to register.

RegistrationForm_v2_IT.java

```
Element email = driver.await(EMAIL); (1)
email.clear(); (2)
email.sendKeys("john@doe.com"); (3)

Element password = driver.await(PASSWORD); (4)
password.clear();
password.sendKeys("secret");
```

1. It uses 3 steps to set user name, `await` is a method from `ExplicitWait`
2. It clears the text in the input field, without this step, the value will be appended to the end of existing text
3. It sets the input field to "[john@doe.com](#)"
4. It uses 3 identical steps to set password

And you can see the steps to set values into either input field or password field are identical, they both have 3 steps, first, call `browser.await(By.name("email"))` to locate the "email" input field, as you learnt from Chapter 13, `await` is a method from `ExplicitWait` and `Browser` inherits it since `Browser` extends `DelegatingWebDriver`, as shown in Figure 13.2 Class Diagram of `DelegatingWebDriver`. Then it clears the text in the input field. Finally it calls the `sendKeys` method of `Element` to set the value. This code looks almost same as Listing 3.2. What we did in Chapter 13 only improves the stability of the code, but haven't improved the cleanliness of the code yet.

There is still code duplication. The code is more verbose than necessary. What different are only the name of the field and value to be set. It is desirable to treat these steps as a unit of work.

Besides input, there are other elements used in HTML forms, which are amongst the most complex standard HTML elements. Let us look at a `radio` button as another example.

Unlike other form elements, a `radio` button will share its name with other elements within the same form. This means that we must find all the elements that match it using one locator, and then iterate over them to find which one is selected or the one we want to select.

Using terms in WebDriver, when you want to see which radio button is selected, you need do the following 3 steps, 1. Locating a list of `WebElement`. 2. Making sure it is not empty. 3. Looping through it to find the element with attribute `checked` being `true`.

This code example demonstrate how to check which choice is checked for "Channel" and "Frequency" on *Radio Button* form (Figure [Registration Form](#))

[NaiveRadioIT.java](#)

```

@Test
public void conact() throws Exception {
    List<WebElement> radiobuttons =
        browser.await(CONTACT); (1)

    assert radiobuttons.size() >= 2; (2)

    for (WebElement e : radiobuttons) { (3)
        if (Boolean.valueOf(e.getAttribute("checked"))) { (4)
            value = e.getAttribute("value"); (5)
            break; (6)
        }
    }

    assertEquals("email", value); (7)
}

```

1. Find a list of elements with the name "contact"
2. Make sure it has at least two elements, otherwise it is not a proper use of radio buttons
3. Loop through the radio buttons

4. Find the one with the attribute "checked"
5. Get the value of radio button found from above
6. Break the loop, this is a good practice
7. You can see the checked item is "email"

From the above examples, we can see it is necessary to refactor the common logic into methods to reduce duplicated code and provide clean interfaces to interact with form elements. We are going to introduce the following techniques to make tasks to interact with form elements easier and robust.

Encapsulating form operations to reduce code complexity

This technique will look at reducing the work needed to test pages containing forms.

From these two examples we demonstrated previously, you can see, there are duplicated code related to form operations, such as entering text and selecting radio buttons, as well as selecting drop-down and ticking checkboxes. As a professional developer, you may want to reduce the total number of lines of code you have to write to complete forms.

It's useful to encapsulate the locating and keying operations into one unit of work: find the element using a given locator, and then interact with the element. This interaction might be to type a value into the input field by entering text, clicking checkboxes, radio buttons, or selecting items from drop-down lists.

Text and password inputs and textarea

Adding a `setInputText` method to the `Browser` class to handle the text field input:

Browser.java

```
public void setInputText(Supplier<By> by, String text) {    (4)
    Element element = await(by);    (1)
    element.clear();                (2)
    element.sendKeys(text);         (3)
}
```

1. Find the element using `await` method with a given locator `by`
2. Clear the text of that element

3. Set the value into the input field

4. This method is to set the input field found using `by` to given `text`

We define the following enum constants in `Name` which implements `Supplier<By>`.

```
EMAIL("email"),
PASSWORD("password"),
```

The earlier code can be rewritten in the following more concise form.

[RegistrationForm_v4_IT.java](#)

```
driver.setInputText(EMAIL "john@doe.com"); (1)
driver.setInputText(PASSWORD, "secret");
```

1. One line of code replaced 3 lines of code

Currently, `setInputText` only accepts strings. You might want to accept other types of data, such as numbers. You can do this by changing the method to accept any object, and then use the object's `toString` method to convert it to a string.

```
public void setInputText(Supplier<By> by, Object text) {
    Element element = await(by);
    element.clear();
    element.sendKeys(text.toString());
}
```

And you can just call the method with the parameter to indicate the name of the field to set and the value to be set, and the value can be any object and you don't need to explicitly convert it into string.

```
int value = 5;
...
browser.setInputText(By.name("amount"), value);
```

Please compare it with the following code which convert integer value into a string explicitly before passing it as the parameter,

```
browser.setInputText(By.name("amount"), String.valueOf(value));
```

You can also add a method to read the input's value and another method to convert the text to upper case,

Browser.java

```
public String getInputText(Supplier<By> by) {
    return await(by).getAttribute("value");
}
```

You can see how simple it became to set values into input or password fields, this method can be used to set value to a text area HTML element as well.

Radio buttons

We can add a `getRadio(By by)` method into `Browser` class to handle the request to read the value from a radio group to see which one is selected, the logic can be copied from Listing [NaiveRadioIT.java](#). Now this `getRadio` can be used to serve that purpose and don't need to repeat the same code again in other tests,

Browser.java

```
public String getRadio(Supplier<By> by) {
    List<WebElement> radiobuttons = findElements(by);      (1)

    assert radiobuttons.size() >= 2;

    for (WebElement e : radiobuttons) {
        if (Boolean.valueOf(e.getAttribute("checked"))) {
            return e.getAttribute("value");
        }
    }
    return null;
}
```

1. We copied logic from `NaiveRadioIT` here

And we can rewrite the test in Listing [NaiveRadioIT.java](#) as following, you can see how simple it became,

RadioIT.java

```
@Test
public void concat() throws Exception { (2)
    assertEquals("email", browser.getRadio(CONTACT)); (1)
}
```

1. `Name.CONACT` is a `Supplier<By>` enum constant

2. Now the test body is just one line of code

Naturally, we'll want a method that will checked the desired element:

Browser.java

```
public void setRadio(Supplier<By> by, String value) {
    List<WebElement> radiobuttons = findElements(by); (1)

    assert radiobuttons.size() >= 2;

    for (WebElement e : radiobuttons) {
        if (value.equals(e.getAttribute("value"))) { (2)
            e.click(); (3)
            return;
        }
    }
    throw new IllegalArgumentException(
        "unable to find element with value " + value); (4)
}
```

1. Find a list of elements with the same locator.

2. Filter those radio buttons so we only have the ones with the correct value.

3. Click it and return.

4. If the elements was not found, throw an exception.

This leads to only one line of code to set the value for each `radio` button group.

Without using the framework, you need to repeat Listing `Browser.java` for each `radio` group, which results in a lengthy test that is very difficult to maintain.

Setting radio buttons

```
browser.setRadio(Name.CONACT, "phone");
browser.setRadio(Name.FREQUENCY, "weekly"); (2)

browser.setRadio(Name.FREQUENCY, "monthly"); (1)
```

1. This code will get an exception since "monthly" isn't a choice in the radio buttons and it is same if you don't use the framework

2. `Name.FREQUENCY` is a `Supplier<By>` enum constant

These methods help make it easier to read or select a value from `radio` button group. Now, let's look at checkboxes.

Checkbox

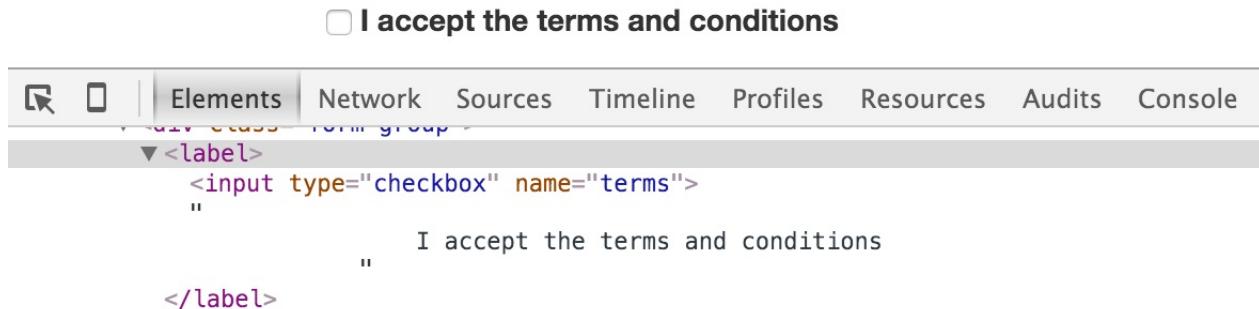


Figure 3. A Checkbox and "Inspect Element" view

Similarly, we can reduce the complexity of checking checkboxes by adding a method to the `Browser` class,

Browser.java

```

public void setCheckboxValue(Supplier<By> by, boolean value) {
    Element checkbox = await(by);
    if (checkbox.isSelected() != value) { (1)
        checkbox.click();
    }
}

```

1. The condition is used to determine whether to do a click. If the checkbox is already checked and you want to set it to true, there will be no need to click.

Why do we add these two methods to `Browser`? Why don't we just use the `click` method from `WebElement`? The framework will allow us to reduce the amount of code needed to reliably click checkboxes. Imagine that, if you don't have these two methods, you need to have the condition check in each place you need to check or uncheck a checkbox. Since whether you need to do a click depends on whether the checkbox is checked or not.

How to use the method

```

browser.setCheckboxValue(Name.TERMS, (3)
                        true) (1)

browser.setCheckboxValue(Name.TERMS, false) (2)

```

1. check the checkbox

2. uncheck the checkbox

3. `Name.TERMS` is a `Supplier<By>` enum constant

Some readers may ask, "why don't you have a method called `checkCheckbox` to check the checkbox and a method called `uncheckCheckbox` to uncheck it?" Normally you will use a variable to hold a boolean value, and use it to decide whether to check or uncheck the checkbox. If you have those two methods, you have to have code with conditional statement.

Suboptimal API design

```
if (toCheck) {                                (1)
    browser.checkCheckbox(Name.TERMS)
} else {
    browser.uncheckCheckbox(Name.TERMS)
}
```

1. You need to have a condition statement to decide to check or uncheck

But `setCheckboxValue` is just one method call. It is clear the design we chose is terser and more robust, you can see the following code using `setCheckboxValue` is very straight forward. It saves you from adding `if` statement in the test code.

```
browser.setCheckboxValue(Name.TERMS), toCheck)
```

That's the reason we only have one method `setCheckboxValue`, not having `checkCheckbox` and `uncheckCheckbox` and think this design is better.

Also, we can add a method to get the checked value:

Browser.java

```
public boolean isChecked(Supplier<By> by) {
    return await(by).isSelected();
}
```

Again, the interaction of checkboxes is simplified to one method call and the details are encapsulated into those methods and developers can just use that method to set to and read from checkboxes. This method greatly improves the productivity of the interaction with checkbox element.

Now let us look at another common element, Select.

Select and multiple select

The Selenium support library provides a `Select` class to handle drop-down operations. But while this works well with static HTML element, it assumes the `option` elements are embedded inside the `select` element as per figure [A Checkbox and "Inspect Element" view](#) below.

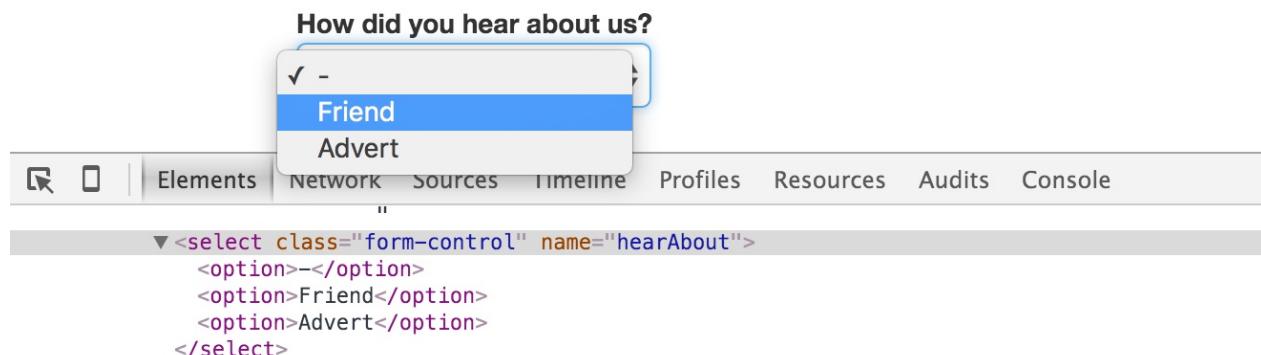


Figure 4. A select and "Inspect Element" view

In web applications built using AJAX, the select maybe empty when clicked on, and then populated asynchronously using JavaScript. the `Select` class does not work for these cases. But, we can add a framework method to handle this scenario. Using a `WebDriverWait` we click on the select and then wait until there are options to choose from:

Browser.java

```
public Select getSelect(Supplier<By> by) {
    final Element element = await(by); (1)
    await(new Predicate<ExplicitWait>() { (2)
        @Override
        public boolean test(ExplicitWait driver) {
            element.click();
            return !element.findElements(TagName.OPTION).isEmpty(); (4)
        }
    });
    return new Select(element);
}

public void selectByVisibleText(Supplier<By> by, String ... values) { (3)
    for (String v: values) {
        getSelect(by).selectByVisibleText(v);
    }
}
```

1. Find the select element.

2. Wait for options to be populated.
3. This can be used to select one or more options
4. `TagName.OPTION` is a `Supplier<By>` enum constant

Replace anonymous inner class with Lambda Expression

If you use Java 8, you can use lambda expression in above code, as in [Browser.java](#)

```
public Select getSelectLambda(Supplier<By> by) {
    Element element = await(by);
    until((ExplicitWait driver) -> {
        element.click();
        return !element.findElements(By.tagName("option"))
            .isEmpty();
    });
    return new Select(element);
}
```

Also, in Java 8, keyword `final` is longer needed here but it is still a final variable

Click and DoubleClick Mouse Action

Besides normal form elements, we can also encapsulate mouse action into `Browser`.

Let us revisit Listing 3.11 [MouseInputIT.java](#)

```
WebElement submitButton = driver.findElement(BUTTON);
new Actions(driver)
    .doubleClick(submitButton)
    .perform();
```

We can move this logic of locating an element and creating an `Action` object and doubleClick it into `Browser` class as `doubleClick(by)` method.

[Browser.java](#)

```
public void doubleClick(Supplier<By> by) { (1)
    Element element = await(by);
    new Actions(delegate).doubleClick(element).perform();
}
```

1. Now you can just call `browser.doubleClick(by)`

And then Listing 3.11 [MouseInputIT.java](#) can be rewritten as following, which is cleaner and concise.

MouseInputIT.java

```
driver.doubleClick(BUTTON);
```

Here is an updated class diagram of `Browser` and you can see these methods are in `Browser` class,

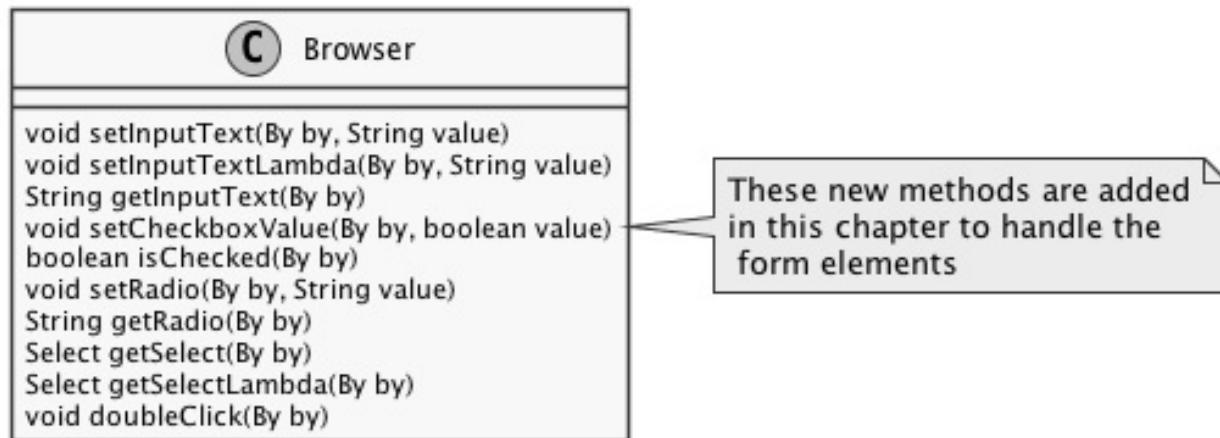


Figure 5. Class diagram of `Browser` and its super classes and interfaces

And this table illustrates what method we can use to change the value or read the value for each HTML element,

Table 2. Methods in `Browser` for form element manipulation

HTML Element	Set value	Read
<code>input type="text"</code>	<code>setInputText</code>	<code>getInputText</code>
<code>input type="password"</code>	<code>setInputText</code>	<code>getInputText</code>
<code>input type="email"</code>	<code>setInputText</code>	<code>getInputText</code>
<code>textarea</code>	<code>setInputText</code>	<code>getInputText</code>
<code>input type="checkbox"</code>	<code>setCheckboxValue</code>	<code>isChecked</code>
<code>input type="radio"</code>	<code>setRadioValue</code>	<code>getRadioValue</code>
<code>select</code>	<code>selectByVisibleText</code>	<code>getSelect(by).getSelectedOptions</code>
<code>input type="button"</code>	<code>doubleClick , click</code>	<code>getInputText</code>
<code>button</code>	<code>doubleClick , click</code>	<code>getText</code>

And the old registration form test can be written as following,

RegistrationForm_v4_IT.java

```

driver.setInputText(EMAIL, "john@doe.com");
driver.setInputText(PASSWORD, "secret");
driver.selectByVisibleText(HEAR_ABOUT, "Friend");      (1)
driver.setRadio(CONTACT, "email");
driver.selectByVisibleText(INTEREST, "Movies", "Music"); (2)
driver.setInputText(TELLUS, "---");
driver.setCheckboxValue(TERMS, true);
driver.click(BUTTON);

```

1. Select a value on single select

2. Select multiple values on a multiple select

You can see it is very clean, but it may not still be clean if the fields become more and more. We will talk about this in this chapter how to handle the page with many input fields.

You've seen some examples of how you can add useful methods to your automation framework. These will reduce the amount of code in your tests. You've also see how you can reduce the number of issues you have by adding error correction code into your framework.

Until now, all form related operations can only be accessed from `Browser`, what if we want to have those methods available from `Element` as well?

It is quite simple, just move those methods into an interface and have both `Browser` and `Element` implement that interface!

Here is the class diagram which contains `FormElements`. You can see those methods are moved from `Browser` class and now become available for `Element`. They are helpful in understanding some classes introduced later.

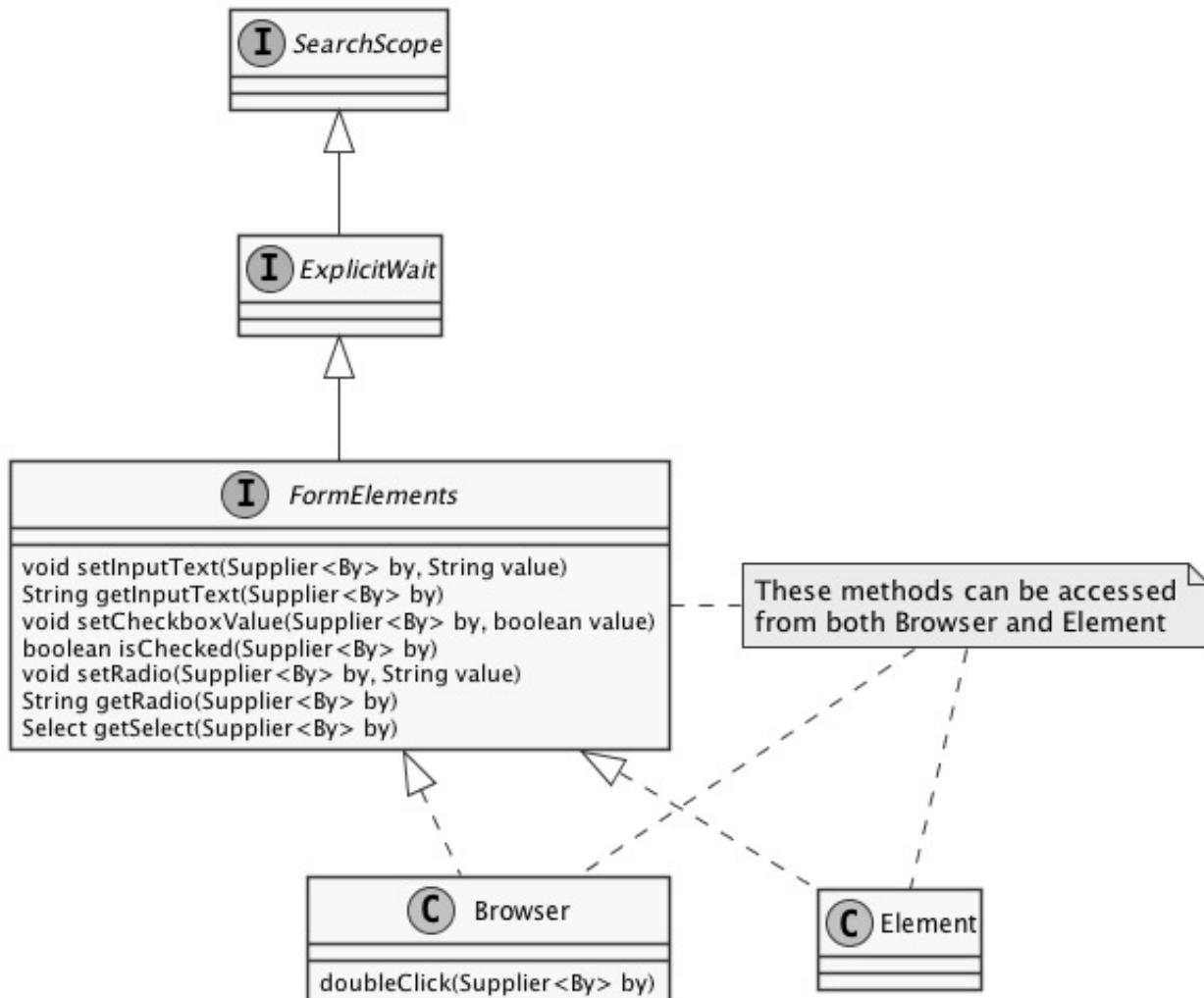


Figure 6. Class diagram of `FormElements`

Of course you can move those methods into `DelegatingSearchContext` and they will still be available from both `Browser` and `Element`. But it is more cohesive to have them in their own place, a `FormElements` interface. This way, it is more compliant with [Interface segregation principle](#) [1]

Interface segregation principle

The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.[1] ISP splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces.[2] ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five SOLID principles of Object-Oriented Design.

[https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))

Next, we'll look at how we can create a retry mechanism that will help in situations where a page changes while you are testing it.

Handling flakiness with a retry

If you have worked on web automation long enough, you will have known that sometimes the test could fail for no clear reason. 98% of time it passes, but fails for the 2% of time. You can't really debug it since it is difficult to reproduce.

For example, we have created a framework which takes a screenshot whenever a button or link is clicked, so we knew what the input values had been set before submitting the form. we found strange behavior with the `sendKeys` method call of `WebElement`. When the test failed, we could see the input field was blank, which means the `sendKeys` method didn't execute correctly, but it also didn't give a stack trace to indicate a failure. So it failed silently and if we hadn't recorded the screenshot before the button was clicked, we wouldn't have noticed this behavior.

Billing Information

First Name:	Sanjay	City:		Credit Card Type:
Last Name:	Rao	may not be empty	American Express	
Street Address 1:	29 Rolling Brook Dr	State:	New_Jersey	Expiry Year:
Street Address 2:		Zip:	08820	May (5)
		Country:	United States	2017

Figure 7. Input without value even the logic is executed

For example, we browsed screenshots for the failed test and we found this screenshot. It says "may not be empty" to one of the fields, but there is logic to set a value to that field in the test script. To make think more difficult, this test doesn't always fail. It just fails occasionally without any clue. And we are sure WebDriver has located that field, otherwise it would throw a `NoSuchElementException`, so we can logically deduct that the `sendKeys` method doesn't already work. But there is no easy way for us to find out why it doesn't work sometimes since when we spend time debugging the test, it always works. Maybe due to the delay caused by the pausing of the program, it gives browser sometime to do things differently than running on the CI server.

Now we are going to show you how to handle this kind of problem by introducing the following technique.

encapsulating retry

Sometimes you may find that an operation on an element does not complete as you expect. You type into an input field, but then you find that the input does not have the text you expect. This technique will show you how to encapsulate a retry mechanism that checks that the action has completed correctly before continuing.

Tests fail as not enough time has passed for the page to be in the expected state, as stated in the beginning of this section, the `sendKeys` method doesn't work as expected but we can't figure out what caused problem.

Create a class that encapsulates retry.

WebDriver cannot tell you when a page has changed. Instead, if the page is not as you expect then you must poll the page until it is. WebDriver provides a built in way to wait for elements to be in a certain state, but this does not easily allow you to click or type as part of that action.

Add a retry mechanism into the logic of entering text into input fields. If the value of the input field is not the same as the value it tries to set, try to do it again until the maximum number of attempts has been reached.

To achieve this goal, we need to create an interface `Attemptable` with only one method `attempt`, so we can provide an implementation for what needs to be attempted inside that method body. If you already use Java 8, you may already know that you can use a lambda expression to provide that implementation. If you still use Java 7 or below, you have to use an anonymous inner class in the place of the Java 8 lambdas express.

[Attemptable.java](#)

```
public interface Attemptable {  
    void attempt() throws Exception;  
}
```

And we can pass the implementation of `Attemptable` interface as a parameter into the `attempt` method of a `Retry` class. The `Retry` class has a constructor which specifies,

1. How many times it will retry
2. What the interval between each retry, this is achieved by the second and third parameters

And in the `attempt` method of the `Retry` class, it simply executes the `attempt` method of the `Attemptable` and repeats it if any exception is caught, until the maximum count is reached.

[Retry.java](#)

```
public class Retry {  
    private final long interval;  
    private final TimeUnit unit;  
    private long count;  
  
    public Retry(int count, int interval, TimeUnit unit) {...} (1)  
  
    public void attempt(Attemptable attemptable) {  
        for (int i = 0; i < count; i++) {  
            try {  
                attemptable.attempt(); (2)  
                return;  
            } catch (Exception e) {  
                if (i == count - 1) { (3)  
                    throw new IllegalStateException(e);  
                }  
            }  
            try {  
                unit.sleep(interval); (4)  
            } catch (InterruptedException e) {  
                throw new IllegalStateException(e);  
            }  
        }  
    }  
}
```

1. You can specify how many times it retries and the interval between each retry.
2. Attempt the task, and return if successful.
3. If you have run out of attempts, throw the exception.
4. Otherwise, wait before trying again.

You can see the relationship between `Retry` and `Attemptable` in the following class diagram.

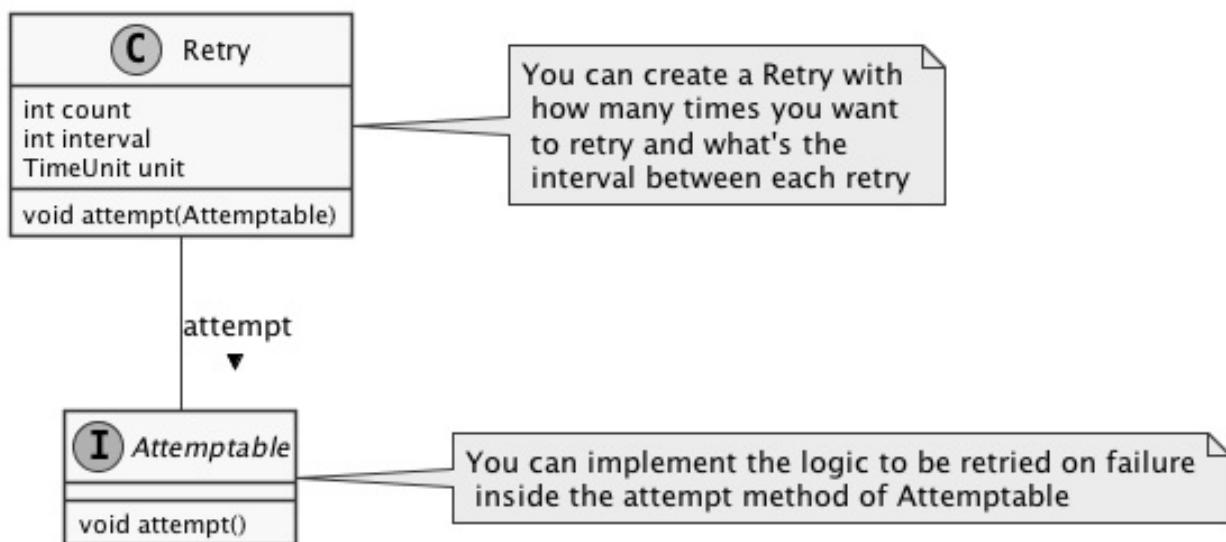


Figure 8. Class diagram of Retry

The `setInputText` method of the `Browser` class can be rewritten to retry until the value is what we were attempting to set it to:

Browser.java

```

public void setInputText(Supplier<By> by, String value) throws Exception {
    Retry retry = new Retry(5, 1, TimeUnit.SECONDS); (1)

    retry.attempt(
        new Attemptable() { (2)
            @Override
            public void attempt() throws Exception {
                Element element = findElement(by); (3)
                element.clear();
                element.sendKeys(value); (4)
                assert value. element.getAttribute("value")); (5)
            }
        }
    );
}
  
```

1. A `Retry` instance is created to retry 5 times at 1 second intervals.
2. Use an anonymous inner class to specify what needs to be retried.
3. Locate the input field.
4. Set the text.
5. Assert the text input is successful, if not we'll retry.

We can use a sequence diagram to illustrate the interaction of these participants,

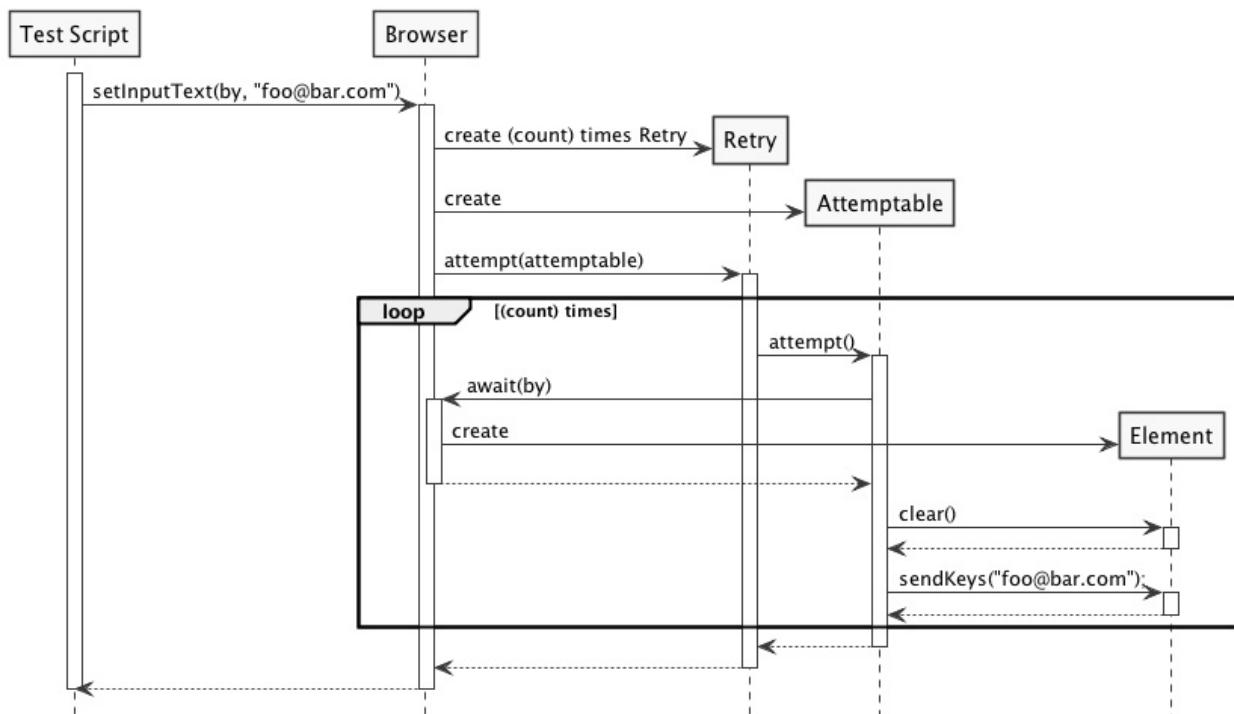


Figure 9. Sequence diagram of `setInputText` using `Retry`

Replace anonymous inner class with Lambda Expression

If you use Java 8, you don't need anonymous inner classes in your code, here is part of the example written using lambda expression, [Browser.java](#)

```

retry.attempt(() -> {
    Element element = findElement(by);
    element.clear();
    element.sendKeys(value);
    assertEquals(value, element.getAttribute("value"));
});

```

We will show you more examples in Lambda Expression later

We've only shown how to apply this to inputs, by this could easily be applied to any element. For example, a select that only loads when you click on it, as discussed earlier in the chapter.

Some readers may ask, why don't you use `WebDriverWait`? It's because, there is no problem of finding the element and the `sendKeys` method call executes without any problem, but sometimes the text is not entered into the text field on web page. This causes the test passes most of times but fails sometimes. The exact reason is still unknown. `Retry` solves this problem nicely. So we use this pragmatic approach rather than spending rest of our life to figure out what's wrong with the `sendKeys` method call.

If you ask us to give any suggestions for choosing how long to wait or how many times to retry, we don't have exact number to tell you, please start from 1 second internal and maximum 5 times as illustrated in the code example. And if that still causes problem, maybe you can increase the retry times or waiting time or combination of both. This is not rocket science, it is more experimental.

Next, we are going to have a look how to manage a page with a lot of elements.

Grouping elements for clarity and maintainability

Please start the web application we provide with the book, and go to this url, <http://localhost:8080/bookstore/cart>, you will see a Shopping Cart page with following form among the other forms.

Other Information

Redemption code:

Billing email:

- Send order messages to this address
- would you like to rate this merchant?

Mailing option:

- Weekly newsletter--New books, updates, news, and special offers.
- Deal of the Day--These amazing special offers last just 24 hours!
- Both
- No promotional mailers. I will still receive updates on my MEAPs and other books.
- Keep me on the lists I'm already on.

Comments:

Figure 10. Other Information Form

As you can see from the screenshot of *Other Information Form* illustrated in Figure [Other Information Form](#), there are many fields, including input fields of "Redemption code", "Billing email", checkboxes of "Send order messages to this address" and "would you like to rate this merchant?" and so on. If we expose all the fields as individual

elements on the page object, the page object would be a very big class with many methods. And it is difficult to understand and maintain. So we introduce the next technique to ease the maintenance of page with many fields.

Organizing elements into logical groups

This next technique will show you how to simplify more complex page objects by breaking them down into smaller parts using logical groups.

On a page with many elements, a page object has become hard to understand because it is large.

In software development, there is a technique called **Delegate Pattern** [2], instead of having one class that does everything, it breaks out some helper classes to manage the details. The master class just delegates the work to its helper classes. It is widely used to reduce the complexity of the code. We are going to create page objects by composing objects together by applying this principle.

Billing Information

First Name:	City:	Credit Card Type:
<input type="text"/>	<input type="text"/>	<input type="text"/>
Last Name:	State:	Credit Card Number:
<input type="text"/>	<input type="text"/>	<input type="text"/>
Street Address 1:	Zip:	Expiry Month:
<input type="text"/>	<input type="text"/>	<input type="text"/>
Street Address 2:	Country:	Expiry Year:
<input type="text"/>	<input type="text"/>	<input type="text"/>

Figure 11. Billing Information Form

On some pages, there are many many elements that you need to interact with. If we expose all the elements via a page object, it will have many many methods. Lets consider the forms illustrated in Figure [Billing Information Form](#) and [Other Information Form](#), which are on the same page of bookstore application. To complete this form, you need complete over twenty fields, which could result in a page object with over twenty fields. Rather than have a single complex page object, this class could be broken up into two classes, one for the "billing information", and one for the "other information".

For now, let us use the fields on *Other Information* form (Figure [Other Information Form](#)) and write a `ShoppingCartPage` class to modify the value of those fields. We can see that it has the following fields:

- An input for a coupon/promotional code.
- An input for an email address.
- Two check boxes: one to receive emails about the order, and a second one to receive emails to allow the buyer to rate the merchant.
- A set of radio buttons to choose mailing options.
- An input for comments.

Replace anonymous inner class with Lambda Expression

If you use Java 8, you can rewrite the for loop using Stream API, it is not necessarily cleaner than before, [MailingOption.java](#)

```
public static MailingOption fromLambda(String string) {  
    Optional<MailingOption> first = Arrays.stream(values())  
        .filter((o) -> string.equals(o.string))  
        .findFirst();  
    if (first.isPresent()) {  
        return first.get();  
    }  
    throw new IllegalArgumentException(  
        "Can't find an enum with this string " + string);  
}
```

If you are interested in learning more of Stream API, here are Java Docs and tutorial from Oracle,

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>

<https://docs.oracle.com/javase/tutorial/collections/streams/>

Just like what we describe in the beginning of this section, if we put every fields into this class as individual element, the class will be very lengthy,

[NaiveShoppingCartPage.java](#)

```
public class NaiveShoppingCartPage {  
  
    private final Browser browser;  
  
    public void setCoupon(String coupon) throws Exception {  
        browser.setInputText(By.id("gc-redemption-code"), coupon);  
    }  
  
    public String getCoupon() {  
        return browser.getInputText(By.id("gc-redemption-code"));  
    }  
    ... (1)  
}
```

1. The rest of the methods are omitted, otherwise this listing will cover two pages

And the test is lengthy too,

[NaiveShoppingCartPageIT.java](#)

```
@Test  
public void weShouldBeAbleToCompleteOtherInformationIndividually() {  
    page.setComment("no comments"); (1)  
    page.setCoupon("no code"); (2)  
    page.setEmail("joe@email.com");  
    page.setRating(true);  
    page.setSendOrderMessages(true);  
  
    assertEquals("no comments", page.getComment());  
    assertEquals("no code", page.getCoupon());  
    assertEquals("joe@email.com", page.getEmail());  
    assertTrue(page.isSendRatingEmail());  
    assertTrue(page.isSendOrderMessages());  
}
```

1. All these fields need to be set and read individually, imagine this page are used in many tests, how much duplicate code
2. `page` is an instance of `NaiveShoppingCartPage`

Remember, right now it is just for *Other Information* only. There are other forms on this page as well, With all the other fields for *Billing Address* and *Credit Card Information* added to the same class, the `NaiveShoppingCartPage` class will be even longer and the test will be longer too.

We are going to show you a different way of doing things. What we can do is to create a Java Bean class, `OtherInformation`,

OtherInformation.java

```
public class OtherInformation extends DomainBase {  
    public final String couponCode;      (3)  
    public final String email;  
    public final boolean sendOrdersToEmail;  
    public final boolean sendRatingEmail;  
    public final String mailingOption;  
    public final String comments;  
  
    public OtherInformation(String couponCode,...) { (2)  
        this.couponCode = couponCode;  
        ... (1)  
    }  
}
```

1. Just assign the parameters to instance variables.
2. The parameters are same as the instance variables
3. Since all instance variables are final, so no getters and setters are needed

Java Bean?

As a Java developer, you probably already noticed that this class is not a Java Bean. That's right. It is not a Java Bean, this class is immutable. So we can expose all fields as public final variables. You don't need any getter or setter methods. To create the instance of the `OtherInformation`, just call its constructor. This is not the approach of Java Bean, which allows change after creation. The instance of this class can't be modified once it is created. It fits perfectly for us to prepare for test data.

Here is the code to create instance of `OtherInformation` class

ShoppingCartPageIT.java

```
new OtherInformation(  
    "no code",  
    "joe@email.com",  
    true,  
    true,  
    "Weekly newsletter--New books, updates, news, and special offers",  
    "no comments");
```

1. Construct an instance of `OtherInformation`.

You can access `sendRatingEmail` directly read this public immutable field. But you can't modify it. It saves from having many getter methods in `OtherInformation` class.

```
other.sendRatingEmail;
```

Then we can use this in an `OtherInformationForm` class as the method parameter,

OtherInformationForm.java

```
public class OtherInformationForm {  
  
    public OtherInformation getOtherInformation() {...}  
  
    public void setOtherInformation(OtherInformation info) { (1)  
        browser.setInputText(COUPON_CODE, info.couponCode);  
        browser.setInputText(BILLING_EMAIL, info.email);  
        browser.setInputText(COMMENTS, info.comments);  
        browser.setCheckboxValue(CONFIRM_EMAIL, info.sendOrdersToEmail);  
        browser.setCheckboxValue(RATINGS, info.sendRatingEmail); (2)  
        browser.setRadio(MAILING_OPTION, info.mailingOption);  
    }  
}
```

1. `OtherInformation` is used as a parameter here
2. Access `sendRatingEmail` directly using public immutable field.

These two classes split the data and the operations into two classes. Then they can be used in the `ShoppingCartPage` class. Instead of having all the fields as separate setter methods, there is just one setter and one getter method on the `ShoppingCartPage` class.

ShoppingCartPage.java

```
public class ShoppingCartPage {  
  
    private final OtherInformationForm otherInformationForm;  
  
    public void setOtherInformation(OtherInformation otherInformation) {  
        otherInformationForm.setOtherInformation(otherInformation); (1)  
    }  
  
    public OtherInformation getOtherInformation() {  
        return otherInformationForm.getOtherInformation(); (2)  
    }  
}
```

1. It delegates the call to `otherInformationForm` variable's setter method

2. It delegates the call to `otherInformationForm` variable's getter method

And the test can be as simple as this,

ShoppingCartPageIT.java

```
@Test  
public void weShouldBeAbleToCompleteOtherInformation() {  
    page.setOtherInformation(info);  
    assertEquals(info, page.getOtherInformation());  
}
```

1. `page` is an instance of `ShoppingCartPage` class and we use one method call to set other information.

Please compare this test with only 2 lines of code with Listing [NaiveShoppingCartPageIT.java](#) which has 10 lines of code and you can see it improves the readability of test.

You can see that splitting the code into small logical groups that we have made the code simpler and more cohesive. Instead of having the `ShoppingCartPage` class to handle all the individual form elements on *Other Information* form, it only handles `OtherInformationForm` class and leaves all the details to the `OtherInformationForm` class.

Summary

- Encapsulate the behavior related to different elements into their own classes to reduce duplicated code,
- Simplify complex operations into methods for commonly used elements on forms such as text inputs, checkboxes, radio buttons and select drop-downs.
- Retry using a retry mechanism to make tests more robust and less prone to failing.
- Organize elements into logical groups rather than exposing them as individual fields to make your page object classes easier to understand and maintain.

In the next chapter you will learn how to apply the techniques learnt prior to this chapter and use them to build an automation for a series of pages to automate a page flow which is essential in testing a web based application.

1. https://en.wikipedia.org/wiki/Interface_segregation_principle
2. https://en.wikipedia.org/wiki/Delegation_pattern

Chapter 15: Automating a page flow

This chapter covers

- Manually navigating the page flow to understand what to be automated
- Creating domain classes to represent important data for the page flow
- Creating page classes to represent important pages in the page flow
- Automating the entire page flow using data for major page flows

So far in this book we have shown ideas and techniques that we hope will deal with problems you might encounter when you automate and test specific parts of web page. When writing a test, you'll often find yourself a test scenario that mean you need to work with several pages in a series of operations where you are locating, interacting and verifying. We call the act of automating a sequence of operations that span multiple pages **page flow**. Understanding page flow concepts, and being able to apply them, will help you become effective and efficient when testing web applications.

This chapter will cover manually navigating the pages, so you can plan out how a test should be written. We'll look at how to create classes that represent the contents of a page, and introduce a technique for automatically retrying operations when problems occur.

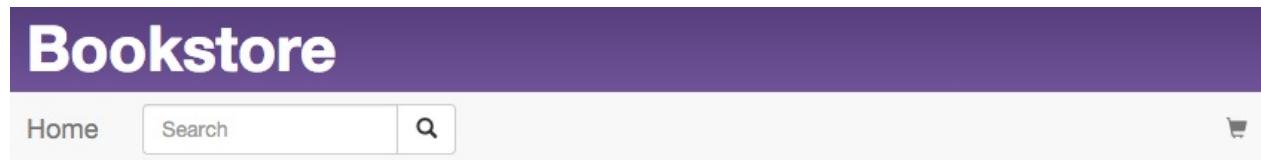
At the end of this chapter you will have learnt how to easily and competently automate and test a page flow.

Manually navigate the page flow to understand what to be automated

On shopping sites, users normally need to go through a series of pages to make a purchase. For example, searching on a search page, selecting the product on product page, setting a quantity on shopping cart page, entering shipping and payment information on shipping and payment pages and finally making a decision to complete the transaction. The web site will also display a confirmation page to let the users know whether the transaction is processed successfully or not. As the quality control engineer for this kind of web site, your job requires writing quality automation scripts

to make sure the application can handle navigation and data entry to fulfill the purchase processing. In order to write automation scripts, first you need to know how to navigate the page flow manually and define a sequence of steps which can be automated using Selenium WebDriver.

Let us start by visiting this famous bookstore at <http://localhost:8080/bookstore/>.



BookStore

Welcome! Please search for books, or go to your cart.

Figure 1. Bookstore Homepage

We are going to go to home page, search for a book called "Selenium WebDriver Book" and click the link on search results to load that book. Then click "add to cart" button to go to shopping cart page and enter required information to make a purchase. We will enter wrong information so the transaction won't go through. (But if you really like the book, you can give correct information and make a purchase.) The purchase flow can be illustrated by this diagram in Figure [Page Flow](#)

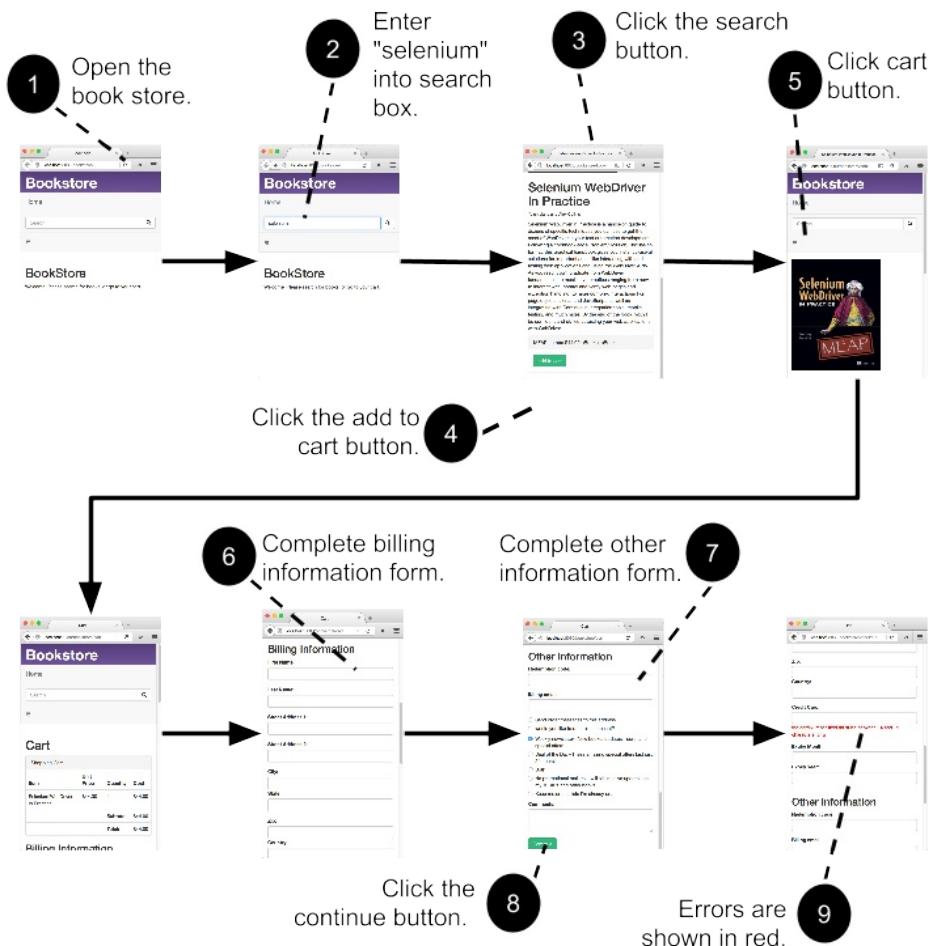


Figure 2. Page Flow

In general, here are the steps to complete the purchase:

- Enter <http://localhost:8080/bookstore/> in the address bar and hit enter, land on the bookstore's home page as illustrated by Figure [Bookstore Homepage](#).
- Type "Selenium WebDriver Book" into the search box and click the search button and get some search results.
- Click the first link on the page to go to the book details page.
- Click "add to cart" button to add the book the the shopping cart.
- Click the cart image button and it will take you to shopping cart page with these 3 forms.
- Complete billing information form
- Complete other information form
- Click the continue button
- And if you don't complete the form correctly, you will see some error messages displayed in red color

In the next section, we are going to show you how to automate the page navigation and data entry tasks to accomplish a purchase flow, using the techniques we learnt from early chapters. Also, we are going to refactor the framework classes to expose a cleaner interface and enhance its robustness by adding an exception recovery mechanism.

Creating domain classes for the page flow

In last chapter, when we introduced the technique to split the page into multiple forms and introduced `OtherInformation` and `OtherInformationForm` class, and use an instance of `OtherInformation` to hold the data and act as the parameter for the `setOtherInformation` method of `OtherInformationForm`.

Create domain classes to represent reference data and form data

When working on automation, we always need to enter data on the form and those data need to be validated by the web application before the page can transition to the next page. So it is critical to know what data can be used to enter the form. And those data may be used over and over again.

Normally, people just used string to hold data, but string is not a good candidate for structured or frequently used data. For example, the names of the all countries in the world and all the states in United States. If we use string to hold those data, it is not well organized and people may created different variables scattered across application.

Create domain package to hold all domain classes. So developers have a common place to store useful design ideas. For this particular bookstore application, since we need to ship the book to a place, so we need to create a class to hold address information, and it can be used for both billing and shipping address,

[Address.java](#)

```

public class Address {
    public final String firstName;      (1)
    public final String lastName;
    public final String street1;
    public final String street2;
    public final String city;
    public final UnitedStates state;
    public final String zipcode;
    public final Countries country;

    public Address(String firstName,...){ (2)
        this.firstName = firstName;     (3)
        ...
    }
}

```

1. Since all instance variables are final, so no getters and setters are needed
2. Constructor parameters are the same as those variables
3. Use the parameters to populate the values in the instance of Address

We can use string as the type for `country`, but it is more desirable to introduce an enum for that purpose. When you use string, people can create many variables in the test to hold those information, For example, one developer may add one variable in one class, but string is not type safe, if you accidentally modify a string, adding or removing a character, the compiler won't be able to notice that mistake. While if all country names are in one enum, it will be more cohesive and people don't need to define those variables in the classes they work on, they can just refer to the enum constants. And if it is changed accidentally, the compiler will tell you that it is not valid.

[Countries.java](#)

```

public enum Countries {
    Australia,           (1)
    China,
    India,
    United_Kingdom,
    United_States,
    ... (2)
}

```

1. Not all capitalized

2. Some methods omitted here

Same is `UnitedStates`, you can define all the states as constants inside `UnitedStates` enum,

[UnitedStates.java](#)

```
public enum UnitedStates {
    ...          (1)
    New_Jersey,
    New_York,
    ...
}
```

1. All of the states in US

Then to refer US, you just use `Countries.United_States`. And to refer New Jersey, you just use `UnitedStates.New_Jersey`.

Java code convention

You may read from Java code convention that enum should be all capitalized, but this way it is more convenient to use so we don't follow the convention. If we followed the Java Code Convention, we would need to define the enum differently,

```
public enum Countries {
    AUSTRALIA("Australia"),
    CHINA("China"),
    INDIA("India"),
    UNITED_KINDOM("United Kingdom"),
    UNITED_STATES("United States"),
    ...

    String name;

    Countries(String name) {
        this.name = name;
    }
}
```

So there will be more code. We prefer the way we choose, but you can choose to follow Java Code Convention. It is not a topic worth debating.

And `CreditCard` class to hold the data for credit card information.

[CreditCard.java](#)

```
public class CreditCard {  
  
    public final String cardNumber;  
    public final String cardCvv;  
    public final String expirationMonth;  
    public final int expirationYear;  
    public final CreditCardType cardType;  
  
    public CreditCard(...) {...} (1)  
}
```

1. Constructor of the CreditCard class

Where `CreditCardType` class is another enum with some major credit cards,

[CreditCardType.java](#)

```
public enum CreditCardType {  
  
    American_Express,  
    JCB,  
    MasterCard,  
    Visa,  
    Discover;  
    ... (1)  
}
```

1. Some methods omitted here

One thing you may notice is that there are a number of mailing list options. And when we use string as parameter, the code is very verbose, as in Chapter 14,

[ShoppingCartPageIT.java](#)

```
new OtherInformation(  
    "no code",  
    "joe@email.com",  
    true,  
    true,  
    "Weekly newsletter--New books, updates, news, and special offers", (1)  
    "no comments");
```

1. The value of this parameter is extremely long

Since each one of them has a long text name, we can also encapsulate them into an enum [MailingOption.java](#)

```

WEEKLY_NEWSLETTER("Weekly newsletter--" +
    "New books, updates, news, and special offers"),
DEAL_OF_THE_DAY("Deal of the Day--" +
    "These amazing special offers last just 24 hours!"),
BOTH("Both"),
NO_PROMOTION_MAILERS("No promotional mailers. " +
    "I will still receive updates on my MEAPs and other books."),
KEEP_ME("Keep me on the lists I'm already on.");
}

private final String string;

MailingOption(String string) {
    this.string = string;
}

```

We need to define a `toString` method for this enum to return the `string` variable.

```

@Override
public String toString() {
    return string;
}

```

Optionally, we can define a method to resolve the enum instance for the give string. There are not many of them, we just loop through all of the enum constants and find the one with same string.

```

public static MailingOption from(String string) {
    for (MailingOption o : values()) {
        if (o.string.equals(string)) {
            return o;
        }
    }
    throw new IllegalArgumentException(
        "Can't find an enum with this string " + string);
}

```

We already have `otherInformation` in Listing 14.20 [OtherInformation.java](#). Since we added this enum for mailing option, we need to change the type for variable

`mailingOption` from `String` to `MailingOption` in `OtherInformation` class.

OtherInformation.java

```
public class OtherInformation extends DomainBase {
    ...
    public final MailingOption mailingOption;
    ...
}
```

In above code, we changed `mailingOption` from `String` to `MailingOption` enum type.

Now that we created package for domain classes, we can use them for the project. And the parameter passing become easier comparing the code in this chapter to some earlier chapters, due to the application of enum in organizing domain information such as the name of the countries, name of the states and type of the credit cards.

Next, we are going to create some page classes to use these domain classes to manage the data entry on each page of the page flow.

Creating page classes to represent important pages in the page flow

From Figure [Page Flow](#), we can see, the page flow starts from visiting the bookstore's homepage, searching on a book, choosing that book, adding that book to cart, going to cart, entering credit card, billing address and other information and confirm the purchase. We are going to create some classes to be responsible for some of the activities. First, let us create `BookstoreHomepage` class.

Creating a `BookstoreHomepage` class for Bookstore's Homepage

We now have enough information to start, we will create page class `BookstoreHomepage` to represent Bookstore's home page, to be in charge of the step 1, 2, 3 in the page flow diagram. We use Web Developer Tool on the browser to examine the elements we need to locate and manipulate, these activities are as following.

1. Enter book name on the input field with `class` attribute "navbar-search"
2. Click the search button with `class` attribute "btn-default" on an element with `id` "secondary-navbar"

By using enum constants, we can define this `ClassName` enum for all the `By.ByClassName` locators we are going to use to locate the elements,

[ClassName.java](#)

```
SEARCH_INPUT("navbar-search"),      (3)
CART_BUTTON("cart-button"),        (1)
SEARCH_BUTTON("btn-default");      (2)
```

1. Locator for Cart button on home page
2. Locator for Search button on home page
3. Locator for search input field

Add the following constant into `Id` enum for the element containing search button.

```
SECOND_NAVBAR("secondary-navbar"),
```

And we can use this enum without calling `.get()`, as shown in the following `searchBook` method of `BookstoreHomepage` class, which represents the page illustrated in Figure [Bookstore Homepage](#).

[BookstoreHomepage.java](#)

```
public class BookstoreHomepage {

    private Browser browser; (6)

    public BookstoreHomepage(Browser browser) { (1)
        this.browser = browser;
    }

    public void searchBook(String bookname) { (2)
        browser.setInputText(SEARCH_INPUT, bookname); (3)
        browser.await(SECOND_NAVBAR) (5)
            .click(SEARCH_BUTTON); (4)
    }
}
```

1. Inject the `Browser` through constructor, we are going to omit this constructor in the other listings.
2. Search book by its name
3. `Name.SEARCH_INPUT` is another enum

4. `SEARCH_BUTTON` is from listing [ClassName.java](#) and `await` method is `findElement` with `wait`
5. Since `SEARCH_BUTTON` is a class name, it may not be unique on the page, so we find its container first.
6. We are going to omit it in other listings.

After clicking the search button, the browser displays a list of books,

Search results - selenium

Total results: 1

Selenium WebDriver Book

Selenium WebDriver Book is a hands-on guide to dozens of specific techniques you can use to get the most of WebDriver in your test automation development. Following a cookbook-style Problem/Solution/Discussion format, this practical handbook gives you instantly-useful solutions for important areas like interacting with and testing web applications and using the WebDriver APIs. As you read, you'll graduate from WebDriver fundamentals to must-have practices ranging from how to interact with, control and verify web pages and exception handling, to more complex interactions like page objects, alerts, and JavaScript, as well as integrating with Continuous Integration tools, mobile testing, and much more. By the end of the book, you'll be confident and skilled at testing your web applications with WebDriver.

Figure 3. Search Result

And we can create a test to run, it simply creates an instance of `BookstoreHomepage` and calls the `searchBook` method with a book name. The test is very short and explains what the test does, to search a book.

[BookstoreSearchIT.java](#)

```
@Test
public void searchBook() {
    homepage = new BookstoreHomepage(browser);
    homepage.searchBook("Selenium WebDriver Book"); (1)
}
```

1. Search this book

This style is much cleaner than exposing all the details in the test. Imagine we inline the details from `searchBook` method into this test method, the developer who need to understand the purpose of the test need go through a mind mapping before understanding the test. Now it is just one meaningful method name, 'searchBook'.

Now we can add a `BookListPage` to continue the page flow.

Creating `BookListPage` for the page with a list of books

This page has two responsibilities,

1. List of the books in the search result.
2. Choose the book you are looking for and go to book details page by clicking the link with the book name.

We want to simplify it so ignore the first responsibility and only code `BookListPage` to allow `chooseBook`, as in the following class.

`BookListPage.java`

```
public class BookListPage {
    ...
    public void chooseBook(String bookname) {
        browser.click(() -> By.partialLinkText(bookname)); (1)
    }
}
```

1. Since `bookname` is a variable, we can only construct a new `Supplier<By>` using Lambda and pass it to `await`

The `chooseBook` method will click the link in the search results illustrated by Figure [Search Result](#), and the browser will transit into a page with the details of the book.

Whether to define `BookListPage`

We create `BookListPage` in the page flow, purely to illustrate the idea of using page object to represent the web pages in a page flow. On a big project, if we define a page object for each page in the application, we may end up with too many page objects. So it is reasonable to combine some page object into another. For example, since the responsibility of the `BookListPage` is only to provide `chooseBook` method to take the page flow to book detail page, we may not need it at all, just move the method body of `chooseBook` into the `searchBook` method of `BookstoreHomepage` class so let `BookstoreHomepage` go directly to `BookPage`.

```
public class BookstoreHomepage {  
  
    public void searchBook(String bookname) {  
        browser.setInputText(SEARCH_INPUT, bookname);  
        browser.await(SECOND_NAVBAR).click(SEARCH_BUTTON);  
        browser.click(() -> By.partialLinkText(bookname));  
    }  
}
```

So we no longer need `BookListPage` under this change so it can be deleted. This approach is commonly used in test automation.

But we still need to define this `BookListPage` if we are required to check the correctness of the search result.

And now we can create a `BookPage` class in the page flow.

Creating a `BookPage` class for Book page

We can create a `BookPage` class to represent the responsibilities on book page, to simulate buyer clicking "add to cart" and "go to cart" buttons to take the book and go to shopping cart page. So it has two methods,

1. `addToCart` - find all elements whose `tagName` is `input` and find the first one with value attribute "add to cart" and click it, in short, click the "add to cart" button.
2. `gotoCart` - find top nav bar and click the cart image button.

[BookPage.java](#)

```

public class BookPage {
    ...
    public void addToCart() {
        browser.findElements(INPUT)      (1)
            .filter(
                e -> e.getValue().equals("add to cart") (6)
            ) (2)
            .findFirst()           (3)
            .get()                 (4)
            .click();
    }

    public void gotoCart() {
        browser.await(TOP_NAV).click(CART_BUTTON); (5)
    }
}

```

1. Find all input fields
2. Find the ones with value "add to cart", which is a Stream
3. Get the first one from the results in Stream, which is a Optional
4. Get the value from the Optional
5. Since CART_BUTTON is class name, so it may not be unique, we find its container first
6. This lambda means if the input element has "add to cart" as attribute return true

Once buyer clicks the shopping cart image button to top of the page, it takes the buyer to shopping cart page, which we will define next.

Creating a `ShoppingCartPage` class for Shopping cart page

In Chapter 14, we already designed a `ShoppingCartPage` class with one `OtherInformationForm` variable. In this chapter, since we need to automate the entire page flow, we will add some new variables to the page to populate the all forms on the page. Those forms are

1. Credit card form, in charge of credit card information.
2. Billing address form, in charge of billing address.

3. Other information form, in charge of the other information.

And we are going to create one class for each form and add as instance variable to `ShoppingCartPage`.

An instance of `CreditCardForm` class

Here is the credit cart form section on the page.

Credit Card Type:

American Express

Credit Card Number:



Expiry Month:

Expiry Year:

Figure 4. Credit Card Form

We can create a `CreditCardForm` class and use its `setCreditCard` method to set the input values represented by an instance of `CreditCard` class to the areas for credit card information as illustrated in Figure [Credit Card Form](#).

[CreditCardForm.java](#)

```
public void setCreditCard(CreditCard card) {
    browser.selectByVisibleText(CARD_TYPE, card.cardType);      (1)
    browser.setInputText(CARD_NUMBER, card.cardNumber);          (2)
    browser.selectByVisibleText(CARD_EXP_MONTH, card.expirationMonth); (3)
    browser.selectByVisibleText(CARD_EXP_YEAR, card.expirationYear); (4)
    browser.setInputText(CARD_CVV, card.cardCvv);                (5)
}
```

1. Select "Credit Card Type"

2. Type "Credit Card Number"
3. Select "Expiry Month"
4. Select "Expiry Year"
5. Type "Card Verification Number"

And create a `BillingAddressForm` to enter the billing address.

An instance of BillingAddressForm

This is billing address section on the shopping cart page,

Billing Information

First Name:	City:
<input type="text"/>	<input type="text"/>
Last Name:	State:
<input type="text"/>	<input type="text"/>
Street Address 1:	Zip:
<input type="text"/>	<input type="text"/>
Street Address 2:	Country:
<input type="text"/>	<input type="text"/>

Figure 5. Billing Address

We use `setBillingAddress` method from `BillingAddressForm` class to set the input values of `Address` to the areas for billing address information as illustrated in Figure [Billing Address](#)

[BillingAddressForm.java](#)

```
public void setBillingAddress(Address address) {    (1)
    browser.setInputText(BILLING_FIRST_NAME, address.firstName);   (2)
    browser.setInputText(BILLING_LAST_NAME, address.lastName);
    browser.setInputText(BILLING_ADDRESS1, address.street1);
    browser.setInputText(BILLING_ADDRESS2, address.street2);
    browser.setInputText(BILLING_CITY, address.city);
    browser.setInputText(BILLING_STATE, address.state);
    browser.setInputText(BILLING_ZIP, address.zipcode);
    browser.selectByVisibleText(BILLING_COUNTRY, address.country);
}
```

1. This is a billing address
2. All the first parameters inside this method are `Supplier<By>` enum constants It is used to type or select these fields on the form, "First Name", "Last Name", "Street Address 1", "Street Address 2", "City", "State", "Zip" and "Country".

Of course, the other information we introduced in last chapter.

An Instance of `OtherInformationForm`

Other Information form is used for collecting coupon code, billing email and comments, allowing buyer to check the checkboxes for "Send order messages to this address" and "would you like to rate this merchant", as well as click the radio button for mailing option in the following Figure.

Other Information

Redemption code:



Billing email:



- Send order messages to this address
- would you like to rate this merchant?

Mailing option:

- Weekly newsletter--New books, updates, news, and special offers.
- Deal of the Day--These amazing special offers last just 24 hours!
- Both
- No promotional mailers. I will still receive updates on my MEAPs and other books.
- Keep me on the lists I'm already on.

Comments:



Figure 6. Other Information

We use `setOtherInformation` method of `OtherInformationForm` class to set the input values of `OtherInformation` to the areas for other information as illustrated in Figure [Other Information](#).

OtherInformationForm.java

```
public void setOtherInformation(OtherInformation info) { (1)
    browser.setInputText(COUPON_CODE, info.couponCode); (2)
    browser.setInputText(BILLING_EMAIL, info.email);
    browser.setInputText(COMMENTS, info.comments);
    browser.setCheckboxValue(CONFIRM_EMAIL, info.sendOrdersToEmail);
    browser.setCheckboxValue(RATINGS, info.sendRatingEmail);
    browser.setRadio(MAILING_OPTION, info.mailingOption);
}
```

1. Set the value on other information section of the page.
2. All the first parameters inside this method are `Supplier<By>` enum constants

Then we can see how these form classes are used in the page class.

And putting things together in `ShoppingCartPage`

And we add these variables to `ShoppingCartPage` class we created in Chapter 14, and now it becomes,

ShoppingCartPage.java

```
public class ShoppingCartPage {  
  
    private final BillingAddressForm billingAddressForm;  
    private final CreditCardForm creditCardForm;  
    private final OtherInformationForm otherInformationForm;  
    ...  
    public void setOtherInformation(OtherInformation otherInformation) {...}  
  
    public void setBillingAddress(Address address) {...}  
  
    public void setCreditCard(CreditCard card) {  
        creditCardForm.setCreditCard(card);  
    }  
  
    public void confirm() { (1)  
        browser.click(CONFIRM); (2)  
    }  
}
```

1. Since `continue` is a Java key word, so we call this method `continues`
2. We can click Continue button by passing `CONTINUE` enum constant

And it is now ready be used for taking inputs. It represents everything in this url, <http://localhost:8080/bookstore/cart>. Once buyer completes all the mandatory information on the form and an order is placed, the bookstore displays a confirmation page with the order number, last page in the page flow.

Creating `ConfirmationPage` class to represent the confirmation page.

The confirmation page displays an order number, so the page class will have this responsibility.

Thank you for your order

Your order number is #00008.



Figure 7. Confirmation page

And we can see the order number "#00008" is on a `span` with `class` attribute "order-number" on a container with `id` attribute `orderNumber`, so we can find the container using `By.id("orderNumber")` first and narrow down the search using `By.className("order-number")` and put the logic inside the `getOrderNumber` method of `ConfirmationPage` class.

ConfirmationPage.java

```
public class ConfirmationPage {
    ...
    public String getOrderNumber() {
        return browser.await(ORDER_NUMBER_ID) (2)
            .getText(ORDER_NUMBER); (1)
    }
}
```

1. We can get the order number by passing `ORDER_NUMBER` enum constant
2. We find the container using `ORDER_NUMBER_ID` enum constant

Now we complete defining all the page classes in the page flow, let us review them using class diagrams. Because the page is not big enough to have them in one diagram, we use two diagrams to display them. That's the only reason they are in two diagrams.

Here are the class diagrams of `BookstoreHomepage`, `BookListPage` and `BookPage`, they all wrap a `Browser` class and act as facade to `Browser`.

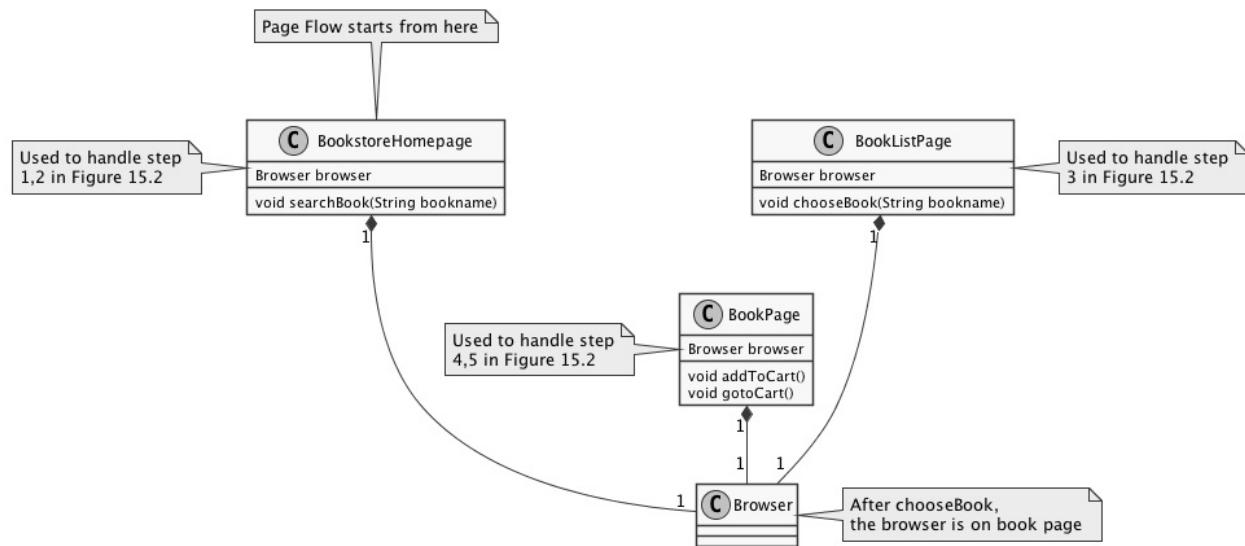


Figure 8. Class diagram of `BookstoreHomepage`

And here are the class diagrams of `ShoppingCartPage`, both of they also wrap `Browser` class.

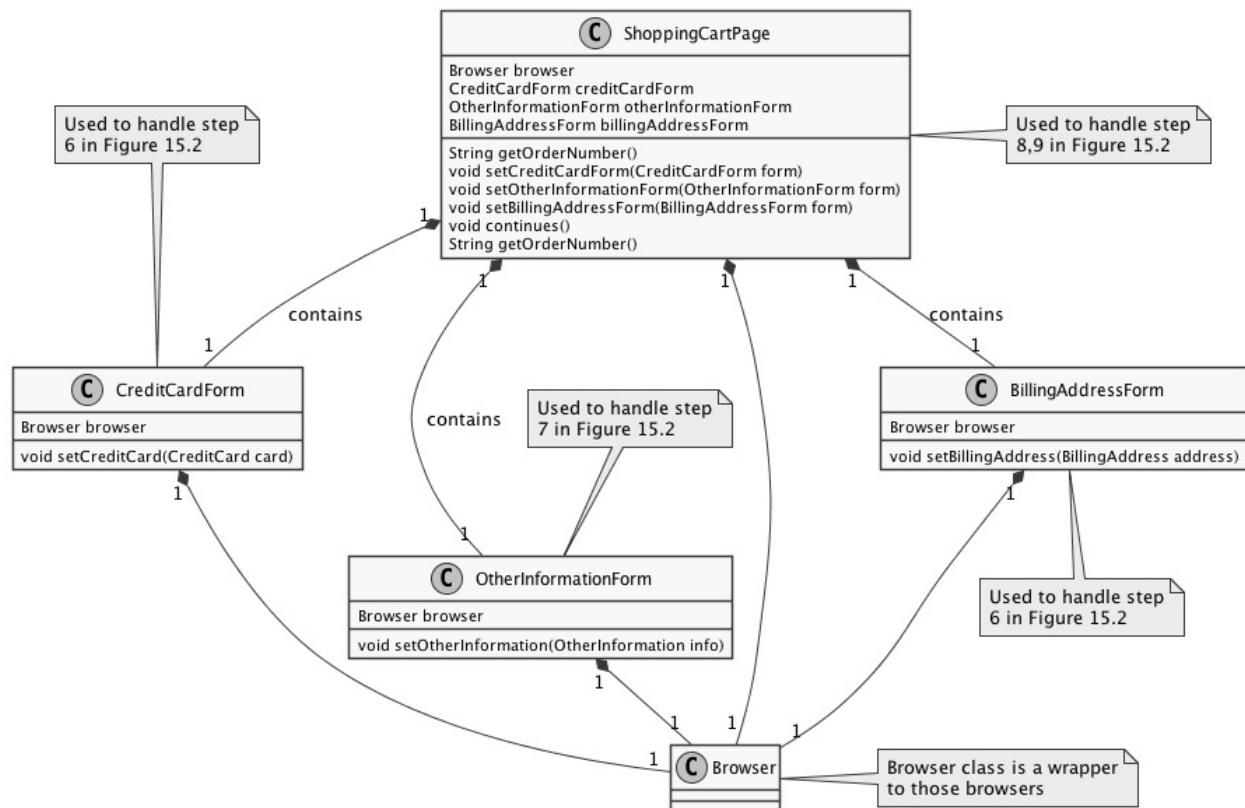


Figure 9. Class diagram of `ShoppingCartPage` and `ConfirmationPage`

These page classes wrap `Browser` class and expose the functionalities as methods to reduce the verbosity in the tests. You can review Chapter 5 for more information about **page object**, besides **page object**, we also introduced Organizing elements into logical groups. `ShoppingCartPage` is the application of that technique and you can see it improves the modularity of the **page object**.

Automating the entire page flow using data for major page flows

When automating a page flow, we normally need to design the test data carefully in order to be able to move from one page to another. Because in the web applications, there are complex validation rules, the data must pass the validation, otherwise, the page will stay on the same page with the error messages. So we need to test both cases, one is to test the page stays when entering invalid data and the page moved to next page when the data pass validation.

Designing test data to fulfill the end to end transaction processing

How to design the test data, the topic alone can be put into one separate book, so we don't want to spend too much time here for this purpose, we simply create some variables in the test so they can be used for the page flow navigation.

First some constants for the expected results,

[BookstoreShoppingIT.java](#)

```
public static final String EXPECTED_ERROR_MESSAGE =
    "The cardNumber must be between 19 and 19 characters long";      (1)
public static final String EXPECTED_ORDER_NUMBER = "#00008";        (2)
```

1. Define an expected error message
2. Define an order number for confirmation

And some variables for the data to be used to fill out the forms in

[BookstoreShoppingIT.java](#)

An instance of `Address`, used to fill out Billing Address using `BillingAddressForm`,

```
private Address billingAddress = new Address( (1)
    "1111 Mountain Dr",
    "14052014",
    "Edison",
    "08820",
    UnitedStates.New_Jersey,
    Countries.United_States,
    "Sanjay",
    "Rao");
```

An instance of `CreditCard`, used to fill out Credit Card for happy path using `CreditCardForm`.

```
private CreditCard creditCard = new CreditCard( (2)
    CreditCardType.MasterCard,
    "4111-1111-1111-1111",
    "123",
    Month.DECEMBER, 2020);
```

Another instance of `CreditCard`, used to fill out Credit Card for invalid input using `CreditCardForm`.

```
private CreditCard invalidCreditCard = new CreditCard( (3)
    CreditCardType.MasterCard,
    "4111-1111-1111",
    "123",
    Month.DECEMBER, 2020);
```

An instance of `OtherInformation`, used to fill out Other Information using `OtherInformationForm`

```
private OtherInformation otherInformation = new OtherInformation( (4)
    "no code",
    "joe@email.com",
    true,
    true,
    MailingOption.WEEKLY_NEWSLETTER,
    "no comments"
);
```

And then writing some tests to automate these two page flows.

Adding tests to run the entire page flows

Now we can finish the test by wiring all the page classes we created in the following test. First let us put common path inside another method so it doesn't repeat.

BookstoreShoppingIT.java

```
@Before (6)
public void addToCartAndSetSomeInformation() {
    BookstoreHomepage homePage = new BookstoreHomepage(browser); (1)
    homePage.searchBook("Selenium WebDriver Book"); (2)

    BookPage bookPage = new BookPage(browser); (3)
    bookPage.addToCart(); (4)
    bookPage.gotoCart(); (5)
}
```

1. Open Bookstore home page
2. Search a book called "Selenium WebDriver Book"
3. Transit to book page on the same browser
4. Click "add to cart" button
5. Click the cart image on top
6. In JUnit, method annotated with `@Before` is executed before each test method

Then the exception flow in this method.

BookstoreShoppingIT.java

```
@Test
public void invalidCreditCard() {
    cartPage = new ShoppingCartPage(browser); (1)
    cartPage.setBillingAddress(billingAddress); (2)
    cartPage.setOtherInformation(otherInformation); (3)
    cartPage.setCreditCard(invalidCreditCard); (4)
    cartPage.confirm(); (5)

    assertEquals(EXPECTED_ERROR_MESSAGE, (6)
        browser.getText(CARDNUMBER_ERROR)); (7)
}
```

1. Transit to shopping cart page on the same browser

2. Enter billing address
3. Enter other information
4. Enter an invalid credit card
5. Click the 'continue' button
6. To check the error message is same as `EXPECTED_ERROR_MESSAGE`
7. `ERROR_MESSAGE` is a locator enum implementing `Supplier<By>`

We can see it runs and page transits from home page to search result page and then book page, shopping cart page and stays on shopping cart page with error messages. Note, we create one page for each page on the page flow. But we don't have to create a class to represent the search result page in Figure [Search Result](#), since we only click a link on it. We can merge the link click into the `searchBook` method. This test case is to automate a purchase flow so it can be omitted.

We also add a test case for successful purchase, you can see by the end an order number has been created. Unlike what we did in Listing [BookstoreShoppingIT.java](#), where we just use the `getText` method of the `browser` object to read the text for error message, in Listing [ConfirmationPage.java](#), we use the `getOrderNumber` method in `ConfirmationPage` for the responsibility of getting order number. And the test uses `getOrderNumber` to get order number/

[BookstoreShoppingIT.java](#)

```

@Test
public void purchaseSuccessful() {
    new ShoppingCartPage(browser) {
        { (3)
            setBillingAddress(billingAddress);
            setOtherInformation(otherInformation);
            setCreditCard(creditCard); (1)
            confirm();
        }
    };
    new ConfirmationPage(browser) {
        { (4)
            assertEquals(EXPECTED_ORDER_NUMBER, this.getOrderNumber()); (2)
        }
    };
}

```

1. To set a valid credit card

2. To check "#00008" which is the value of EXPECTED_ORDER_NUMBER will be displayed on page.
3. To create an instance of an anonymous inner class of ShoppingCartPage and what inside the inner bracket is an anonymous constructor block
4. Please note, this is 2 curly brackets.

The following sequence diagram illustrates this test,



Figure 10. Sequence diagram of the purchase flow.

Anonymous inner class

You may already be familiar with creating an anonymous inner class using an interface, such as

```

new Function<WebElement, WebElement>() {
    @Override
    public WebElement apply(WebElement webElement) {
        return webElement.findElement(
            By.className("ignore-react-onclickoutside"));
    }
}
  
```

You can create anonymous inner class using non final class or abstract class as well.

For example, in Listing 15.19, we created two anonymous inner classes. For `new ConfirmationPage(browser) { ... }`, what the compiler does is to create an anonymous

inner class `BookstoreShoppingIT$2` that extends `ConfirmationPage` class and put the code inside the curly bracket as an anonymous constructor block inside `BookstoreShoppingIT$2` class, as illustrated in the following code snippet,

```
private class BookstoreShoppingIT$2 extends ConfirmationPage {  
    BookstoreShoppingIT$2(Browser browser) {  
        super(browser);  
        assertEquals(EXPECTED_ORDER_NUMBER, this.getOrderNumber());  
    }  
}
```

So the original code will be compiled to code using the anonymous inner class.

Original source code

```
new ConformationPage(browser) { ... }
```

Compiled code.

```
new BookstoreShoppingIT$2(browser);
```

This coding style makes test code more concise. But you don't have to use it if you don't like it. And It is not specific to Selenium WebDriver, so if you like it, you can use in other Java project as well.

Normally we don't include import statement for classes in the book, but we are going to show you all the import statements of this test. It doesn't import any classes or interfaces from Selenium Library, which mean we wrap WebDriver inside page object so the test code even doesn't have knowledge of Selenium WebDriver.

Among the classes we import in this test, these are JUnit classes and static method.

```
import org.junit.Before;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import static org.junit.Assert.assertEquals;
```

These two classes are from Java JDK.

```
import javax.inject.Inject;  
import java.time.Month;
```

These are the domain classes defined for the pages.

```
import swb.ch15pageflow.domain.*;
```

These are all the page classes and a supplier locator

```
import swb.ch15pageflow.pages.*;
import static swb.locators.Id.CARDNUMBER_ERROR;
```

And `Browser` and `BrowserRunner` to start the test.

```
import swb.framework.Browser;
import swb.framework.BrowserRunner;
```

With this design, you don't need to import Selenium WebDriver classes in the test at all. This is a commonly used pattern in enterprise software development, to have an in-framework to prevent the direct access to third party libraries. Either to attach more responsibility or increase the consistency of the application.

Summary

- We always need to navigate from one page to another manually to understand the page flow, which gives us a good idea about what you are going to automate using WebDriver.
- Create domain classes for the form sections on complex page.
- Define different page classes to represent different pages in the page flow.
- Remember you don't need to write one class for each page, especially if the page is just used to transition into another page.
- You can organize page classes in one test method to carry out the navigation and data entry to complete the transaction of buying a book from an online bookstore.

In the next chapter you will learn how to automate the access to a complex table element.

Chapter 16: Examining HTML Tables

This chapter covers

- Extracting heading into a list of strings
- Creating domain classes representing the table header and table row
- Validating table using customized result comparator
- Using Java Generics to remove duplicated code

HTML tables are widely used for tabular data presentation. Normally each row in a table represents the same type of data. For example, in the following table, each row is a person. These columns represent an Id, first name, last name and age. And your task is to validate this table to make sure the data are correct.

People

Id	First Name	Last Name	Age
1	Eve	Jackson	94
2	John	Doe	80
3	Adam	Johnson	67
4	Jill	Smith	50

Figure 1. An Example of An HTML Table

Let's take a look at the HTML behind this table, there is an `id` attribute for the table tag and we can use it to locate the entire table element.

[people-table.html](#)

```

<table class="table table-striped" id="users-table">
  <thead>
    <tr>
      <th>Id</th><th>First Name</th><th>Last Name</th><th>Age</th> (1)
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>1</td><td>Eve</td><td>Jackson</td><td>94</td>
    </tr>
    ...
  </tbody>
</table>

```

1. We assume when we develop a web application with a table, we use `TH` tag for table header.

From the table, how can we locate all the table cells? In Chapter 2 we introduced a technique related to locate a cell on a table using `CssSelector`. And in Chapter 12 we introduced an interface `Table` and provided a `SimpleTable` implementation for you to locate a cell by its row number and column number. We believe the technique in Chapter 12 greatly simplifies your tasks of locating cells with known positions on a table.

That technique is good for locating few elements on a table. If you want to do a thorough examination of the table, it is not enough. For example, if you want to validate the information displayed on table is correct and use the technique we learnt from Chapter 12. Since you can only verify each cell individually, the code will be lengthy.

NaiveTableIT.java

```

assertEquals("Id", table.getHeader(1).getText()); (1)
assertEquals("First Name", table.getHeader(2).getText());
assertEquals("Last Name", table.getHeader(3).getText());
assertEquals("Age", table.getHeader(4).getText());

assertEquals("1", table.getBodyCell(1, 1).getText()); (2)
assertEquals("Eve", table.getBodyCell(1, 2).getText());
assertEquals("Jackson", table.getBodyCell(1, 3).getText());
assertEquals("94", table.getBodyCell(1, 4).getText());
... (3)
}

```

1. Read the header from the table and compare each heading with the expectation
2. Read each cell and compare with the expectation
3. The rest of code is omitted

You can see from above listing, this table only has 4 rows and 4 columns of data, the code is already very long. Imaging a table with 100s of rows.

Also, the test will stop at the first error. And if there is any change in the requirements, the data in the table changes accordingly, the test must be changed. Once you have fixed the first error, it may fail at another place, and on and on. You will be exhausted after you fix all the errors for this test!

The test shown above is written in a common seen style in many organization, it is verbose, and violates the single assertion rule in software test. When data changes, you may need to fixed the error one by one and it may be very time consuming to fix all the errors. We just list it here as an anti-pattern for you to be aware of.

Single assertion rule

Single assertion rule means you should only have one assertion statement in each test. The above code apparently violates that rule and is considered an antipattern. We convert it into a test following single assertion rule, it will be,

```

@Before
public void readTable() {
    browser.get("/people-table.html");
    table = new SimpleTable(browser.await(TABLE));
}

@Test
public void header1ShouldBeld() {
    assertEquals("Id", table.getHeader(1).getText());
}

...
@Test
public void row1Column1ShouldBe1() {
    assertEquals("1", table.getBodyCell(1, 1).getText());
}
...

```

This test looks clean, but its `readTable` method will be executed for each test method. For this table, it is 20 times. And if we use `@BeforeClass` instead of `@Before`, we won't be able to inject the `Browser` needed for the test.

It doesn't need to be this way. In this chapter, we are going to introduce some new techniques related to examining a table. And we are going to show you how to design a general class to handle HTML tables to eliminate the amount of repeated code. When requirement changes, you can fix all the errors on one single run.

By the end of this chapter, you will have learnt how to read data from a table into a list of domain objects and use a customized data analyzer to analyze the data and output a meaningful error report with all differences found between the expected and actual results.

Extracting heading into a list of strings

First, we need to locate the table and its heading. We follow the same locating technique we learned from Chapter 2. We just try to show you how to extract the heading into a list of strings representing each column. By doing this, when we compare the data with the expectation.

We now create a `PersonTable` class to extract the headings,

[PersonTable.java](#)

```
public class PersonTable {  
  
    private final Element table;  
  
    public PersonTable(Element table) {  
        this.table = table;  
    }  
  
    public List<String> getHeaders() {    (3)  
        List<String> headers = new ArrayList<>();  
        for (WebElement th : table.findElements(By.tagName("th"))){    (1)  
            headers.add(th.getText());    (2)  
        }  
        return headers;  
    }  
}
```

1. This logic is based on the assumption we had earlier in the HTML code for the People table.
2. We use all style for loop to process the collection so we use old method to find elements

3. In the latter part of this chapter, we are going to use Java 8 Stream to rewrite this method

Some readers may ask, what about the data in the table? Should we use a list of lists? Or an array of strings to represent the rows in the table? Our answer is, yes, you can use both ways. There is a disadvantage of doing these ways. The expectation is not type safe, so you can put a value which is not a number in the place for `id` and `age`. Those errors can only be detected when you run the tests. In the following example, you made a mistake of using "b" instead of "1", you can only detect this mistake when you run the test.

```
assertEquals("b", table.getBodyCell(1, 1).getText());
```

When you run it, it fails with assertion error.

```
org.junit.ComparisonFailure:  
Expected :b  
Actual   :1  
<Click to see difference>
```

So we are going to show you another way of extracting data. And you can see this kind of error will be caught during compile time. In order to do that, we need to introduce a domain class first.

Creating domain classes representing table rows

We are going to introduce a domain class to represent the people table we are trying to automate, it has id, first name, last name and age as properties.

Creating domain object super class

Instead of using array of strings, we can create an entity class to represent the data on each row. For the particular table example, we will create a `Person` class. Unlike the Java class you might create for the production code, you may not need to provide getters for this class. But it does need at least an `equals` method for comparison and a `toString` method for debugging purposes. Now you can see by overriding the `equals` method in `Person` class, we effectively encapsulate the logic to compare two instance

of `Person` class. We can create one instance and use it as the expectation and read the data row on one HTML table and create another instance, then we can use the `Assert.assertEquals` method from JUnit testing framework. While it will be cumbersome to use JUnit if you just extract the data into list of strings, the expectation data will not be type safe as shown before .

In tests, We need to compare the equality of other domain objects and output debugging messages for developer to analyze the cause of the failure. So we need to provide some customized `equals`, `hashCode` and `toString` method to those classes as well. In order not to repeat those methods in every classes we create, we introduce this `DomainBase` class as the super class for those domain classes, so they don't need to provide these methods if they can use these default ones in `DomainBase` class,

[DomainBase.java](#)

```
public abstract class DomainBase {    (1)

    @Override
    public boolean equals(Object other) {    (2)
        return EqualsBuilder.reflectionEquals(this, other);    (3)
    }

    @Override
    public int hashCode() {    (4)
        return HashCodeBuilder.reflectionHashCode(this);    (5)
    }

    @Override
    public String toString() {    (6)
        return ToStringBuilder.reflectionToString(this);    (7)
    }

}
```

1. We make it abstract so don't want people to create an instance of this class
2. Override the `equals` method from `Object` class
3. Use `EqualsBuilder` to check whether two instances are equal
4. Override the `hashCode` method from `Object` class
5. Use `HashCodeBuilder` to calculate the hash code of this object
6. Override the `toString` method from `Object` class

7. Use `ToStringBuilder` to output the content of this object

Any class extends `DomainBase` inherits these three methods. But if they don't meet your needs, you still can override them in the subclass, we are going to override one method when we create subclasses, `toString` to illustrate this.

Apache Commons Library

We use Apache Commons library here to for those method. If you are not familiar with it, you can find out more from the project website,

<https://commons.apache.org/>

Creating domain class

Now we create `Person` class and it extends `DomainBase` class. Since we are not satisfied with the `toString` method from `DomainBase` class, we override it in `Person` class to output different format.

Person.java

```
public class Person extends DomainBase {
    private final int id;
    private final String firstName;
    private final String lastName;
    private final int age;

    public Person(int id, String firstName, String lastName, int age) {...} (3)

    @Override
    public String toString() { (1)
        return "new Person(" + id +
            ",\\" + firstName + "\",\\" +
            lastName + "\", " + age + ")\\n"; (2)
    }
}
```

1. We can define how to output the person, this is just one example and it is not prescriptive but we do have our reason to do this way
2. The reason `toString` is coded this way is that we can copy from the output of test run and use it to modify the expectation
3. Standard constructor

When you calling the `toString` method of `Person` object, it will output the way how it is created. This can be illustrated in the following test,

PersonTest.java

```
private Person person = new Person(1, "John", "Doe", 44);

@Test
public void testToString() throws Exception {
    assertEquals("new Person(1, \"John\", \"Doe\", 44)\n", person.toString());
    System.out.println(person);
}
```

You can see it prints,

```
new Person(1, "John", "Doe", 44)
```

This output can be copied and used to change the expectation when test fails. We will apply this in next section of this chapter.

Once you create this `Person` domain class, you can use it to map each row in the table into an instance of this class. We will show you this technique in the following section.

Extracting rows into a list of domain objects

We are going to add a **row mapper** as a constant of the `PersonTable` class we introduced earlier in Listing [PersonTable.java](#). The responsibility of row mapper is to map each row on the table to an instance of the `Person` class. So we use the `Function` interface which takes a list of `Element`, where the first element is the first cell on that row, and second element is the second cell, and so on. And, it creates a instance of the `Person` class. We use pre-java 8 style to create an anonymous inner class of `Function`, implements its `apply` method here as a starting point. The code is a bit verbose. We will use Java 8 style later, and the code will be cleaner. So you can compare them.

PersonTable.java

```

public static final Function<List<Element>, Person> MAPPER_NON_JAVA_8    (1)
= new Function<List<Element>, Person>() {          (4)
@Override
public Person apply(List<Element> cells) {      (5)
    return new Person(           (2)
        Integer.parseInt(cells.get(0).getText()), (6)
        cells.get(1).getText(),                  (7)
        cells.get(2).getText(),                  (8)
        Integer.parseInt(cells.get(3).getText()) (3)
    );
}
};

```

1. `MAPPER_NON_JAVA_8` will map the data in each row into a `Person` class
2. Create an instance of `Person` class using those parameters
3. Use the fourth element as the fourth parameter
4. Create anonymous class of `Function<List<Element>, Person>` type
5. Provide implementation on the fly
6. Convert the first element into `Integer` and pass as the first parameter
7. Use the second element as the second parameter
8. Use the third element as the third parameter

And add `getRow` method to extract the table rows using the mapper `MAPPER_NON_JAVA_8`.

PersonTable.java

```

public List<Person> getRows() {      (2)
    List<Person> rows = new ArrayList<>();  (1)

    for (WebElement tr : table.findElement(By.tagName("tbody")) (3)
        .findElements(By.tagName("tr"))) { (4)

        List<Element> cells = new ArrayList<>();
        for (WebElement cell : tr.findElements(By.tagName("td"))) { (5)
            cells.add(new Element(cell)); (6)
        }
        rows.add(MAPPER_NON_JAVA_8.apply(cells)); (7)
    }
    return rows;
}

```

1. Create a list of `Person`
2. `getRow` will return a list of those instances of `Person` class
3. Find `<tbody>` element on the `<table>` element
4. Find all `<tr>` elements inside the `<tbody>` element and loop though it
5. Loop through the list of `<td>` elements found on each `<tr>` element
6. Create an instance of `Element` class and add to a list of `cells`
7. Use `MAPPER_NON_JAVA_8` mapper to convert the list of cells of `Element` into a `Person` object and add to a list of `Person` objects

`PersonTable` class extracts a list of instances of the `Person` class from the HTML table. And we are going to verify the results in the next section.

We can move `MAPPER_NON_JAVA_8` in Listing [PersonTable.java](#) out of `PersonTable` class into its own place, such as an enum constant,

[PersonMapper.java](#)

```
public enum PersonMapper implements Function<List<Element>, Person> {

    MAPPER_NON_JAVA_8 {
        @Override
        public Person apply(List<Element> cells) {
            return new Person(...);
        }
    }
}
```

And you can delete the `MAPPER_NON_JAVA_8` variable from `PersonTable` and just import this enum constant, it works exactly same as before. That way, the test is laid out cleanly with the intention, which is not buried in details.

Once we read the table contents into a list of domain object using mapper, we can validate them using some customized result comparator to generate a report shown the different between the expected results and what is actually displayed on the table. So we can find out all the differences between what should be displayed on table and what is actually displayed on table in a very easy to digest format. It helps the developers to understand the problem better and make it easier to make changes to the failed tests and make them work.

Validating a table using customized result comparator

Test purpose of test is to make sure the application behaves correctly. But it will be nice if the test can help developer to diagnose the problem and make it easier to fix the failed tests.

Next we are going to introduce a technique to make debugging a failed test easier.

Defining Customized Result Comparator to output detailed difference report

Often, people write tests which fail at the first failed assertion, just as illustrated by listing [NaiveTableIT.java](#). There is a fundamental drawback with that approach. When things go wrong, you can only fix one problem on each run since each time you can only see one failed assertion. Thus it can be extremely time consuming to fix failed tests, especially when you need to navigate a couple of pages before making an assertion. Often people blame Selenium for that. It is not a problem with Selenium but with the way the tests are written.

This technique will show you how to make the test easier to debug.

Tests are hard to debug due to limited diagnostic output. And after you fix one problem, you will get another problem. This goes on and on.

One approach is to find all possible errors so we can fix them together.

We are going to create a `PersonTableContents` class to output all the differences between expected result and actual contents. This class has two responsibilities,

- Compare it with an actual result to see whether they are same
- Calculate the difference between the expected and actual result.

The `describeDiff` method of `PersonTableContents` class will find out what is missing in the actual result as well as what in the actual result is not expected. It extends `DomainBase` and inherits the three methods from `DomainBase`.

[PersonTableContents.java](#)

```

public String describeDiff(PersonTableContents other) {    (1)
    return diffHeaders(other)
        + diff(this.rows, other.rows, "expected rows not found: ")    (3)
        + diff(other.rows, this.rows, "unexpected rows appeared: ");    (4)
}

public String diffHeaders(PersonTableContents other) {    (2)
    StringBuilder diff = new StringBuilder();
    if (!headers.equals(other.headers)) {
        diff.append("headers differ ")
            .append(headers).append(" vs ")
            .append(other.headers).append("\n");
    }
    return diff.toString();
}

public <T> String diff(List<T> rows1, List<T> rows2, String s) {    (7)
    List<T> diff = new ArrayList<>(rows1);
    diff.removeAll(rows2);    (5)

    return diff.isEmpty() ? "" : s + diff + "\n";    (6)
}

```

1. This method is used to output the contents so JUnit can compare expected result with actual result.
2. Generate the difference of headers
3. Find out the rows in expected but not in actual
4. Find out the rows in actual but not in expected
5. Removing the elements in the second parameter rows2, the whatever left is the ones only in the first parameters row1
6. Format the message using the third parameter when there is difference
7. This method is parameterized

Then we can modify `PersonTable` class to add the method to create `TableContents`, it just creates an instance of `PersonTableContents` using header and rows,

```

public PersonTableContents getContents() {
    return new PersonTableContents(getHeaders(), getRows());
}

```

And we can write a `PersonTableIT` to verify the contents are correct,

PersonTable_v1_IT.java

```
public static final PersonTableContents OUTDATED_EXPECTED = (4)
    new PersonTableContents(
        Arrays.asList("Id", "First Name", "Last Name", "Age"),
        Arrays.asList(
            new Person(1, "Eve", "Jackson", 94) (1)
            , new Person(2, "John", "Doe", 80) (3)
            , new Person(4, "Jill", "Smith", 50)
            , new Person(5, "Jack", "Clyde", 78) (1)
        )
    );
}

@Test
@Ignore("You can remove this to run it and check the output") (2)
public void testReadFromPersonTableButFailed() {

    PersonTable table = new PersonTable(browser.await(TABLE)); (7)

    PersonTableContents actual = table.getContents(); (5)

    assertEquals(OUTDATED_EXPECTED.describeDiff(actual), OUTDATED_EXPECTED, actual); (6)
}
```

1. This row actually doesn't exist, so the test will fail, let us run it and observe the output
2. You can remove this line to run the test
3. There is a reason why the comma start from the beginning of the second line
4. To create an expected table contents for the test
5. Get the actual table contents from HTML page
6. Compare the actual with the expected and output the difference information
7. Create `PersonTable` using `<table>` element

When you run the test, it will fail and you can observe the following output on the console

The screenshot shows a JUnit test failure report. At the top, it says "1 test failed - 288ms". Below that is the stack trace:

```

java.lang.AssertionError: expected rows not found: [new Person("5","Jack","Clyde",78)
]
unexpected rows appeared: [new Person("3","Adam","Johnson",67)
]
Click to see difference

<1 internal calls>
at org.junit.Assert.failNotEquals(Assert.java:834) <1 internal calls>
at swip.ch16table.v2.person.PersonTableIT.missingExpectedValues(PersonTableIT.java:95) <8 internal calls>
at org.springframework.test.context.junit4.statements.RunBeforeTestMethodCallbacks.evaluate(RunBeforeTestMethodCallbacks.java:73)
at org.springframework.test.context.junit4.statements.RunAfterTestMethodCallbacks.evaluate(RunAfterTestMethodCallbacks.java:82)
at org.junit.rules.ExpectedException$ExpectedExceptionStatement.evaluate(ExpectedException.java:239)
at org.junit.rules.RunRules.evaluate(RunRules.java:20)

```

Figure 2. Failed test with detailed reason for failure.

And after you click the link of `<Click to see difference>`, the following window will be popped up to tell you the difference between expected result and actual result.

The screenshot shows a comparison tool window titled "Comparison Failure". It has two tabs: "Expected (Read-only)" and "Actual (Read-only)". Both tabs show a list of Person objects with their attributes. There are two differences highlighted in green:

Expected (Read-only)	Actual (Read-only)										
[Id, First Name, Last Name, Age], [new Person("1", "Eve", "Jackson", 94), , new Person("2", "John", "Doe", 80), , new Person("4", "Jill", "Smith", 50) , new Person("5", "Jack", "Clyde", 78)]	<table border="1"> <tr><td>1</td><td>[Id, First Name, Last Name, Age], [new Person("1", "Eve", "Jackson", 94)</td></tr> <tr><td>2</td><td>, new Person("2", "John", "Doe", 80)</td></tr> <tr><td>3</td><td>, new Person("3", "Adam", "Johnson", 67)</td></tr> <tr><td>4</td><td>, new Person("4", "Jill", "Smith", 50)</td></tr> <tr><td>5</td><td>]]</td></tr> </table>	1	[Id, First Name, Last Name, Age], [new Person("1", "Eve", "Jackson", 94)	2	, new Person("2", "John", "Doe", 80)	3	, new Person("3", "Adam", "Johnson", 67)	4	, new Person("4", "Jill", "Smith", 50)	5]]
1	[Id, First Name, Last Name, Age], [new Person("1", "Eve", "Jackson", 94)										
2	, new Person("2", "John", "Doe", 80)										
3	, new Person("3", "Adam", "Johnson", 67)										
4	, new Person("4", "Jill", "Smith", 50)										
5]]										

Figure 3. Difference between expected result and actual result.

Also, if you made the same mistake of using `"b"` instead of `1`,

The screenshot shows an IntelliJ IDEA code editor with a yellow tooltip. The tooltip says: "Person() in Person cannot be applied to: Expected Parameters: id: int Actual Arguments: b". A green callout bubble points to the word "b" with the text "Invalid data will cause compilation error".

```

new Person(1, "Eve", "Jackson", 94)

```

it would cause a compilation error, as in the following figure.

The screenshot shows an IntelliJ IDEA code editor with a yellow tooltip. The tooltip says: "Person() in Person cannot be applied to: Expected Parameters: id: int Actual Arguments: b". A green callout bubble points to the word "b" with the text "Invalid data will cause compilation error".

```

@Te
pub
    new Person("b", "Eve", "Jackson", 94)
    new Person("2", "John", "Doe", 80)
    , 67)
    50)

```

Figure 4. Compilation error caused by type mismatch detected by IntelliJ IDEA

All IDEs such as IntelliJ, Eclipse and NetBeans can detect mistake earlier and save you time.

While you work on a project, likely things will change soon and often, you end up changing the test a lot. This technique can help you to improve the productivity of changing the test, as illustrated in Figure [Difference between expected result and actual result.](#), it is obvious to know the difference between the expected results and actual result. With a more detailed test report, you will have all information to fix the test quicker than writing a test which fails at the first assertion failure.

Also we explained why the `toString` method is like following,

```
@Override  
public String toString() {  
    return "new Person(" + id + ", " + firstName + "\", \"  
           lastName + "\", " + points + ")\n";  
}
```

You can see the message in the error report can be copied as code. Once you have verified that the following result is correct,

```
, new Person(3, "Adam", "Johnson", 67)
```

you can copy it to replace the old expectation `OUTDATED_EXPECTED` from Listing [PersonTable_v1_IT.java](#).

```
, new Person(5, "Jack", "Clyde", 78)
```

This can save you some effort from typing the new code in the the test.

Remove duplicated code

In real life project, there will be tables for different data types. For example, not only will you have a table of people, but you will have a table of cities as well, as shown in the following figure. [1]

Cities

Id	City Name	State Name
1	Xian	Shanxi
3	Guangzhou	Guangdong
3	Shaoguan	Guangdong
4	Tianjin	Tianjin
5	Changsha	Huana
6	Shenzhen	Guangzhou
7	Hong Kong	Hong Kong
8	Hangzhou	Zhejiang
9	Singapore	Singapore
9	New York	New York
10	Sydney	New South Wales
11	Dallas	Texas

Figure 5. Another Example of An HTML Table

We are gong to show you a solution without using Java Generics first. You can see, even we use object oriented principle and introduce an abstract class and remove obvious duplicated code, we can't remove the code differ only in types. After you finish reading this section, you will have a clear understanding why sometimes we need to use generics.

To validate this city table, we can follow the same approach we validate people table. Just like we create a `Person` class for people table, we create a `city` class for city table.

[City.java](#)

```

public class City extends DomainBase {
    private final int id;
    private final String name;
    private final String stateName;

    public City(int id, String name, String stateName) {...} (3)

    @Override
    public String toString() { (1)
        return "new City(\"" + id +
               "\",\"" + name + "\",\"" + stateName + "\")\n";
    }
}

```

1. Override the `toString` from `DomainBase` class
2. The purpose of this method is also generating code
3. Standard constructor

And here is the class diagram of `Person` and `City` classes, they both override the `toString` method of `DomainBase` super class to have their own format.

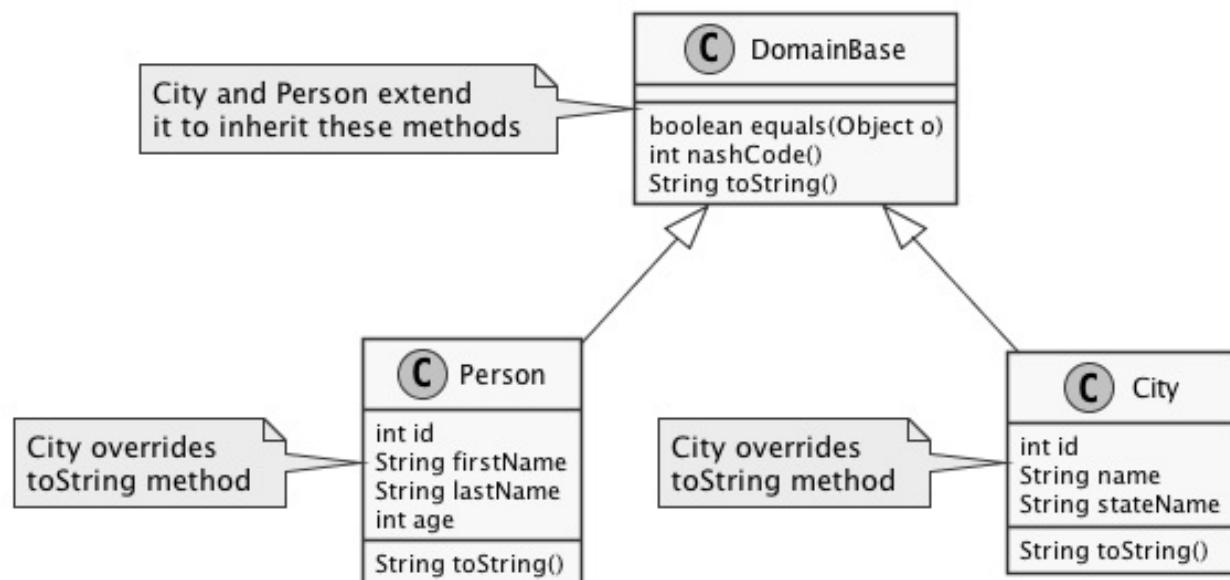


Figure 6. Class diagram of `City` and `Person`

You can create a `cityMapper`, which does not need to be an enum constant as the `PersonMapper`, we can use a lambda expression to create a `public final static` variable in `cityMapper` class instead. We just give an alternative here and we think they both are good design so you can choose one of them.

[CityMapper.java](#)

```

public class CityMapper {

    public final static Function<List<Element>, City> MAPPER_LAMBDA = (1)
        cells -> (2)
            new City( (3)
                Integer.parseInt(cells.get(0).getText()), (4)
                cells.get(1).getText(), (5)
                cells.get(2).getText()); (6)
}

```

1. `MAPPER_LAMBDA` constant to map a list of `Element` objects into a `City` object
2. Use lambda expression to create the `MAPPER_LAMBDA` of type `Function<List<Element>, City>`
3. Create an instance of `City` class using those parameters
4. Convert the first element into `Integer` and pass as the first parameter
5. Use the second element as the second parameter
6. Use the third element as the third parameter

Next we are about to create a `CityTable` class using this mapper, which is very similar to `PersonTable` class. But since the `getHeaders` method is same for both `PersonTable` and `CityTable`, so we are going to introduce a technique to remove the duplicates. Due to the limitation of that technique, the duplicate can only be removed partially.

Using abstract class to partially remove duplicated code

We need to create `CityTable` class to extract the headers and rows information on city table page.

If we create `CityTable` class by copying `PersonTable` class, `getHeaders` method is same for both `PersonTable` and `cityTable`.

we can create an abstract class with `getHeaders` method and delete the one in `PersonTable`.

[AbstractTable.java](#)

```

public abstract class AbstractTable {
    protected Element table;          (1)

    public AbstractTable(Element table) {
        this.table = table;
    }

    public List<String> getHeaders() {      (2)
        List<String> headers = new ArrayList<>();
        for (WebElement th : table.findElements(By.tagName("th"))) {
            headers.add(th.getText());
        }
        return headers;
    }
}

```

1. It needs to be a `protected` variable in order for its subclasses to access it.
2. This method is deleted from `PersonTable` class

Then we can have `CityTable` class extending `AbstractTable` class so we don't need to have the `getHeader` repeating here.

CityTable.java

```

public List<City> getRows() {    (4)
    List<City> rows = new ArrayList<>();    (2)

    for (WebElement tr : table.findElement(By.tagName("tbody"))
        .findElements(By.tagName("tr"))) {

        List<Element> cells = new ArrayList<>();
        for (WebElement cell : tr.findElements(By.tagName("td"))) {
            cells.add(new Element(cell));
        }
        rows.add(MAPPER_LAMBDA.apply(cells));    (5)
    }
    return rows;
}

public CityTableContents getContents() {
    return new CityTableContents(getHeaders(), getRows());    (3)
}

```

1. This for loop statement is almost identical to the one inside the same method in `PersonTable`, with difference in mapper and the type of the `rows` variable

2. It creates a list for `City` objects
3. Create `CityTableContents` object for the content of the table,
4. This method is almost same as the one in `PersonTable` except it returns a `List<City>`
5. Create a `City` object from `cells` and add to `rows`

As well as a `CityTableContents` class, also similar to `PersonTableContents` class. We follow the same approach as the `AbstractTable` class and create an `AnstractTableContents` class and move two methods originally in `PersonTableContents` over, as following,

[AbstractTableContents.java](#)

```
public class AbstractTableContents extends DomainBase {

    private final List<String> headers;

    public AbstractTableContents(List<String> headers) {
        this.headers = headers;
    }

    public String diffHeaders(AbstractTableContents other) {
        StringBuilder diff = new StringBuilder();
        if (!headers.equals(other.headers)) {
            diff.append("headers differ ")
                .append(headers).append(" vs ")
                .append(other.headers).append("\n");
        }
        return diff.toString();
    }

    public <T> String diff(List<T> rows1, List<T> rows2, String s) {
        List<T> diff = new ArrayList<>(rows1);
        diff.removeAll(rows2);

        return diff.isEmpty() ? "" : s + diff + "\n";
    }
}
```

And have `cityTableContents` class extending it, so `diifHeaders` and `diff` methods don't need to be repeated in `CityTableContents` class,

[CityTableContents.java](#)

```

public String describeDiff(CityTableContents other) { (1)
    return diffHeader(other)
        + diff(this.rows, other.rows, "expected rows not found: ")
        + diff(other.rows, this.rows, "unexpected rows appeared: ");
}

```

1. The code in this method is almost same as `PersonTableContents`, the only different is the parameter `CityTableContents`

And you can run the following test to verify the result. You need to comment out the line starting with `@Ignore` annotation to activate this test, otherwise it is ignored by JUnit test framework,

[CityTable_v1_IT.java](#)

```

private static final CityTableContents OUTDATED_EXPECTED =
    new CityTableContents(
        Arrays.asList("Id", "City Name", "State Name"),
        Arrays.asList(
            new City(1, "Xian", "Shanxi")
            , new City(2, "Guangzhou", "Guangdong")
            , new City(3, "Shaoguan", "Guangdong")
            , new City(11, "Dallas", "Texas")
        )
    );
}

@Test
@Ignore("You can remove this to run it and check the output") (1)
public void failedToReadFromTable() {

    CityTable table = new CityTable(browser.await(TABLE));

    CityTableContents actual = table.getContents();

    assertEquals(OUTDATED_EXPECTED.describeDiff(actual), OUTDATED_EXPECTED, actual);
}

```

1. You can delete this line to run this test

This is a failed test and you will have enough information to working on the fix of the tests. The reason we provide failed tests here is to illustrate the benefit of having customized comparison and error reporting in the tests.

You can see, by using inheritance, we can only partially remove the duplicated code, `getHeaders` is in `AbstractTable` class so both `CityTable` and `PersonTable` inherit it from `AbstractTable`. They both have `getRows` method defined individually, even those two methods are quite similar. However, we can't have a single method to handle both table, person and city tables, since `Person` and `City` are too different types.

Here is the class diagram with both sets of classes for city and people table,

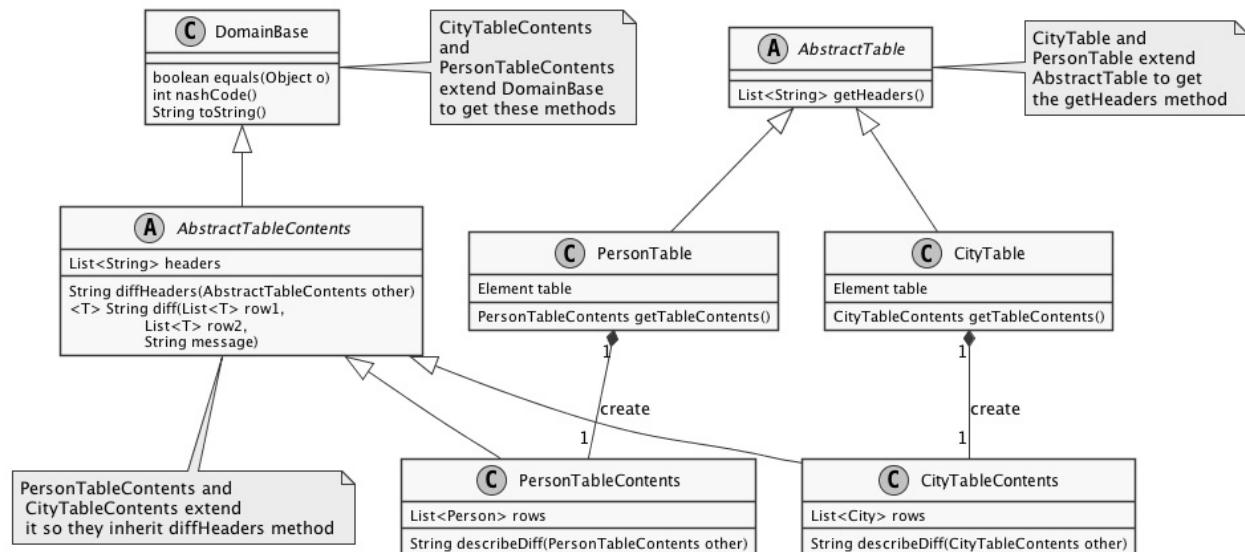


Figure 7. Class diagram of classes for city and people table

Probably you already noticed that there are many similarities between the two sets of the classes and code for `getRows` in two table classes and `describeDiff` in two table contents are not exactly but looks very similar. And we can't use inheritance to remove the duplicates.

The duplicate can be removed. We are going to show you how to use Java Generics to remove the duplicated code. But if you are an experienced developer, probably you will just start with generics. We create these two sets of classes is to educate less experienced developers how to refactor the tests.

Using Java Generics to remove duplicated code

We are going to show you how to use Java Generics to reduce duplicated code.

You noticed that there are many similarities between the two sets of the classes and most codes are duplicated, even you tried to use inheritance but there is some duplicate code which can't be reduced using inheritance.

We will use Generics introduced from Java 5. Instead of creating `PersonTable`, `CityTable` and so on, we just create one `Table<T>` class. And when we want to use it, we pass in a generics parameter to the `Table` class, such as `Table<Person>`, `Table<City>`.

We are going to add a **row mapper** into the constructor of a `Table<T>` class to replace both `PersonTable` and `CityTable` classes. What this row mapper does is to map each row on the table to an instance of the `T` class we pass in. For `Table<Person>`, it will map to a list of `Person`, but for `Table<City>`, it will map to a list of `City`.

Table.java

```
public class Table<T> { (1)

    private final Element table;
    private final Function<List<Element>, T> rowMapper; (2)

    public Table(Element table, Function<List<Element>, T> rowMapper) {
        this.table = table;
        this.rowMapper = rowMapper;
    }

    private Stream<String> getHeaders() {...}

    private Stream<T> getRows() {...} (3)

    public TableContents<T> getContents() { (4)
        return new TableContents<>(getHeaders(), getRows());
    }
}
```

1. `rowMapper` will map the data in each row into a domain class dominated by the type parameter
2. The mapper maps the list of `td` element into type `T`
3. Find all `th` elements and return a `Stream<T>`
4. Create a `TableContents<T>` instance

We use Java 8 feature in this class, and if you are not comfortable with the syntax, please refer the sidebar for more information.

Stream API from Java 8

We use Java 8 Stream API in `getHeader` and `getRows` methods. If you are not familiar with it, you can find out more from Java website,

<https://docs.oracle.com/javase/8/docs/api/index.html>

and click the `java.util.stream` link for that package.

The code is actually used Stream API to map the table row into domain object `T`,

```
private List<String> getHeaders() {
    return table.findElements(TH)
        .map(Element::getText)
        .collect(Collectors.toList());
}

private List<T> getRows() {
    return table.await(TBODY)
        .findElements(TR)
        .map(tr ->
            rowMapper.apply(
                tr.findElements(TD).collect(Collectors.toList())
            )
        )
        .collect(toList());
}
```

If you are not familiar with this style, it is fine. You can use pre-Java 8 style to work with Generics and remove the duplicated code. Generics came in Java 5.

Same as we use `Table<T>` to replace `PersonTable` and `CityTable`, we will use `TableContents<T>` to replace `PersonTableContents` and `CityTableContents`.

TableContents.java

```
public class TableContents<T> extends DomainBase {

    private final List<String> headers;
    private final List<T> rows;

    public TableContents(List<String> headers, List<T> rows) {
        this.headers = headers;
        this.rows = rows;
    }

    public String describeDiff(TableContents<T> other) {...} (2)

    private String diffHeaders(TableContents<T> other) {...} (3)

    private String diff(List<T> rows1, List<T> rows2, String s) {...} (1)
}
```

1. This method is same as the one in `CityTableContents` except it doesn't need the type parameter `<T>` since it is on class level
2. The method body is same as the `describeDiff` method in `PersonTableContents`
3. The method body is same as the `diffHeaders` method in `AbstractTableContents`

To see how can you pass in a mapper to the constructor of `Table` class, and we can use the `MAPPER_LAMBDA` variable in `cityMapper` class from Listing [CityMapper.java](#).

```
public final static Function<List<Element>, City> MAPPER_LAMBDA =
    cells ->
        new City(
            Integer.parseInt(cells.get(0).getText()),
            cells.get(1).getText(),
            cells.get(2).getText());
```

Now we invoke its constructor with an instance of `<table>` element found by calling `browser.await(TABLE)` and `MAPPER_LAMBDA` to construct an instance of `Table<City>`.

```
Table<City> table = new Table<>(browser.await(TABLE), MAPPER_LAMBDA);
```

Or if you use Java 8, you can have the following concise form by using Lambda expression to create a `rowMapper` and use it as the second parameter to create an instance of `Table` class.

[CityTable_v2_IT.java](#)

```
@Test
public void testReadFromTableJava8() {

    Table<City> table = new Table<>(browser.await(TABLE), (1)
        cells -> (2)
            new City(Integer.parseInt(cells.get(0).getText()), (3)
                cells.get(1).getText(),
                cells.get(2).getText())
    );
    TableContents<City> actual = table.getContents();

    assertEquals(EXPECTED.describeDiff(actual), EXPECTED, actual);
}
```

1. The instance of `Element` class representing the HTML table element

2. This is a lambda expression to create an instance of `Function<List<Element>, City>` and use it as the second parameter to create `Table` object
3. The purpose of lambda expression is exactly same as variable `MAPPER_NON_JAVA_8`, to convert a list of `Element` into an instance of `City` class

And you can use the same `Table<T>` class to automate the people table as well, just pass a different mapper as the second parameter, you can use the `MAPPER_NON_JAVA_8` enum from Listing [PersonMapper.java](#),

```
MAPPER_NON_JAVA_8 {  
    @Override  
    public Person apply(List<Element> cells) {  
        return new Person(...);  
    }  
}
```

And we invoke the constructor with an instance of `<table>` element found by calling `browser.await(TABLE)` and `MAPPER_NON_JAVA_8` to construct an instance of `Table<Person>`.

```
Table<Person> table = new Table<>(browser.await(TABLE), MAPPER_NON_JAVA_8);
```

Here are the reduced classes for city and people table, and if you add more domain to the project, you don't need to add new table classes.

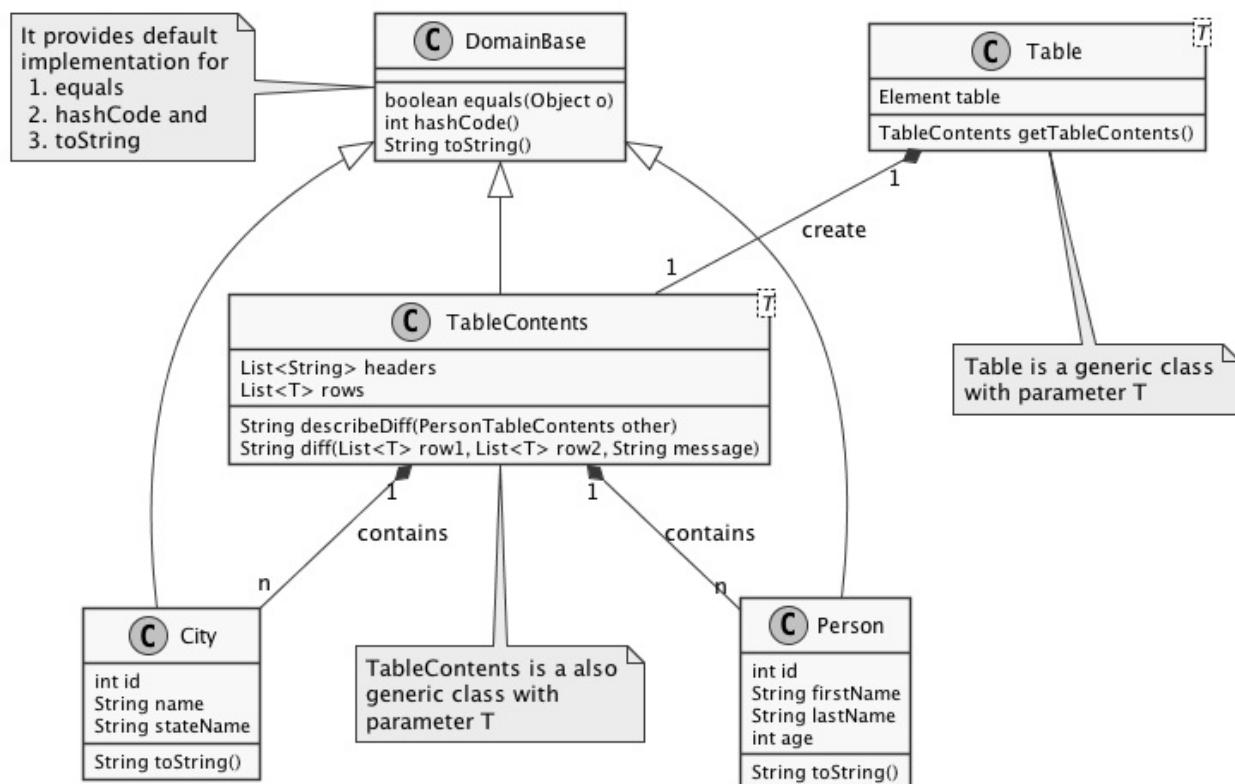


Figure 8. Class diagram of generic classes for city and people table

With the help from Generics, you turn the two abstract classes into two concrete classes with type parameter.

Now you can see, we only need one `Table<T>` class one `TableContents<T>` class to handle HTML tables with different data types, such as `Person` and `City`, and other data type you use in your projects. The different mapping logic is managed by the row mapper classes we create for the data types. Generics is a powerful way to reduce duplication and you will find it useful.

We spent a whole chapter on table because table is a very complex HTML element on web page. This chapter just gives you an idea what can be done when validating a table, but we don't think the `Table` and `TableContents` classes can be applied to any tables out there in the wild so we don't put it under the `framework` package.

When you validate a table, you will definitely find the techniques from this chapter are useful. The customized comparator can also be applied to other kind of tests as well, it is not Selenium WebDriver specific. Generics is also very commonly used in Java programming language.

Summary

- You can identify the properties of table and define a correspondent class as the place holder for the properties
- You can use a general purposed `Table` class to take a function as parameter to locate a table and its rows and columns
- Validate the contents of the table and generate output for all the differences on one run to save the time of running the tests over and over again by using the failure on first error approach.
- Use Generics to remove duplicated code.
- Use `toString` method to generate usable code.

In next two chapters, we are going to automate jQuery datepicker and extract a framework and use it to automate some popular datepickers such ReactJS, Bootstrap, Material-UI and JsDatePick.

1. In case you are wondering why these cities are listed here, they are actually all the cities one of the author have lived in the past.

Chapter 17: Automating jQuery Datepicker

This chapter covers

- Understanding the datepicker
- Automating the jQuery datepicker
- Introducing collaborators to split responsibilities

In Chapter 6, we showed you how to trigger the display of a ReactJS calendar. But we didn't go into details about picking a date. Because it needs an entire chapter to automate a datepicker. So we decided to do it when we have a framework to do the finding and clicking. And now it is time for us to automate it.

The reason datepicker made its place in this book is because it is a commonly used widget on web application, and it is difficult to automate. The reason it is difficult to automate lies on the fact they come out of third party libraries and may not have an easy way for Selenium to locate its elements. Also the logic to pick a date is complex.

By the end of this chapter, you'll learn to automate one datepicker, which will allow you select a date from a datepicker calendar and use that to fill out the date field on a web page.

Understanding datepickers

On web applications, for example, flight booking and hotel booking, users need to enter dates for the flights and hotel. Without any help, users will be confused about what is expected format they should use. That is because when talking about date, we live in a varied world because there are so many date formats used by different countries, to represent April 1st, 2014,

Table 1. Date formats in some countries

Country	Format
Example	USA
MM/dd/YYYY	04/01/2014
China	YYYY-MM-DD
2014-04-01	Australia
dd-MM-YYYY	01-04-2014
UK	dd-MM-YYYY

But on a web page, usually the date field expects only one format for user input. In the past, an instruction about the format is annotated next to the date field to tell users what's the expect format of the date (see figure [A date field with annotation of the format it expects](#)), with the hope it can reduce the chance of users entering invalid dates.



Please enter data in format of MM/dd/YYYY

Figure 1. A date field with annotation of the format it expects

Even with this prompt, it was still possible for you to enter an invalid date, and you'd only know about it when you submitted the form you were filling out. It was annoying.

Also, this approach cluttered the web pages with information only useful for the first time and it soon became nuisance for subsequent visits. As web technology became more advanced, many input assistant libraries were added to make the web page more user friendly. Datepicker is one of them.

What is a datepicker

A datepicker is an input assistance method to help user to select a date and converts the user selection into the format expected by the web pages. The application of datepicker on web page greatly reduced the confusion on the date format, made it impossible to enter invalid dates, and improved user experience. On web pages, all HTML datepickers are built using JavaScript libraries, with or without a calendar icon next to the input field.

Let us take a look of a datepicker built using Material-UI Date Picker, it is associated with an input field on form, you can click it to open an interactive calendar, flip the calendar to a future or past month and choose a day. The date you chose is shown as the input's value.

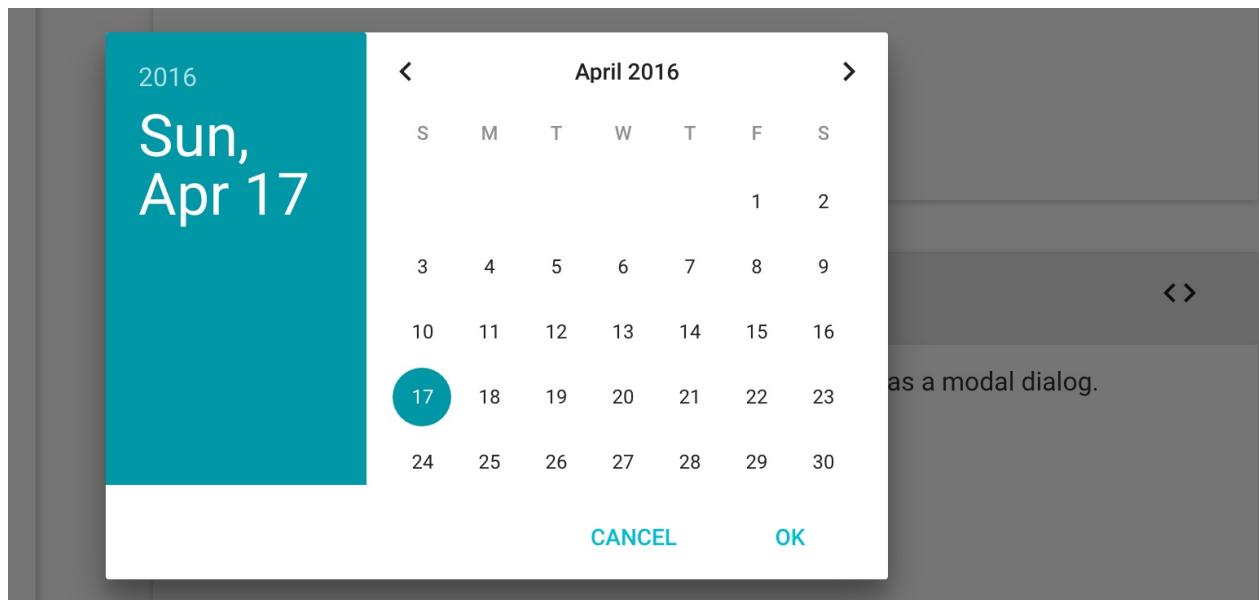


Figure 2. Material-UI Date Picker

Let us say today it is April 17th, 2016 and you want to pick April 1, 2014. Because it is a past date, you need to click previous month arrow < and keep track of the change on the calendar until it is on the month of April 2014. Then you click 1 on the calendar to choose the day. Depending on the implementation of the calendar, it may not close, so we need to take further action to close the calendar in order to move on. For this particular example, we need to click OK to close the calendar.

Since they are built using JavaScript, the timing of the elements appear on the page is unpredictable and the structure of datepicker is very complex. Also some calendar, such as Material-UI, uses **CSS Transition** to provide a transition effect when people flip the calendar. All these web techniques complicate web automation. Many people struggled to get it working and the solutions people posted on internet are very inconsistent.

Despite the difficulties we are facing, datepickers can still be automated. Let us start by breaking down the steps taken in automating a datepicker. Same as automating other web elements, we need to,

1. Locate the important elements on a datepicker.
2. Trigger the display of the calendar

3. Read the calendar title to find out the current month and year and decide the direction to flip the calendar.
4. Based on the direction, click either previous or next month button to flip the calendar and,
5. Choose a day when reaching the target month.

First let us have a look what elements a datepicker has and how can we locate them.

Identifying the elements of a datepicker

As the first step to automate date picking, you need to locate the trigger element to display the calendar. The trigger can be an input field or a calendar icon. When you click the input field, a calendar pops up, so we define the input field as a "Trigger", which triggers the display of the calendar, as shown below,

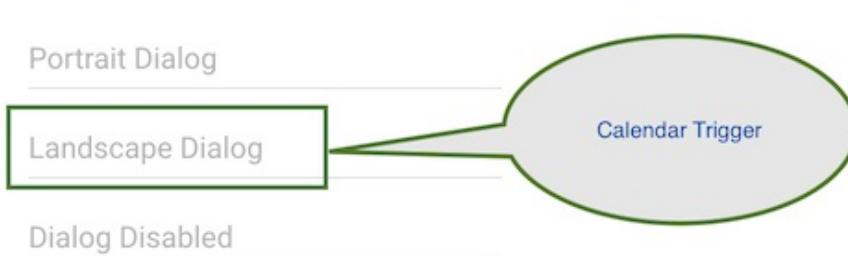


Figure 3. Datepicker Calendar Trigger

Then you need to locate other calendar elements. For Material-UI datepicker, these elements are annotated on the following figure,

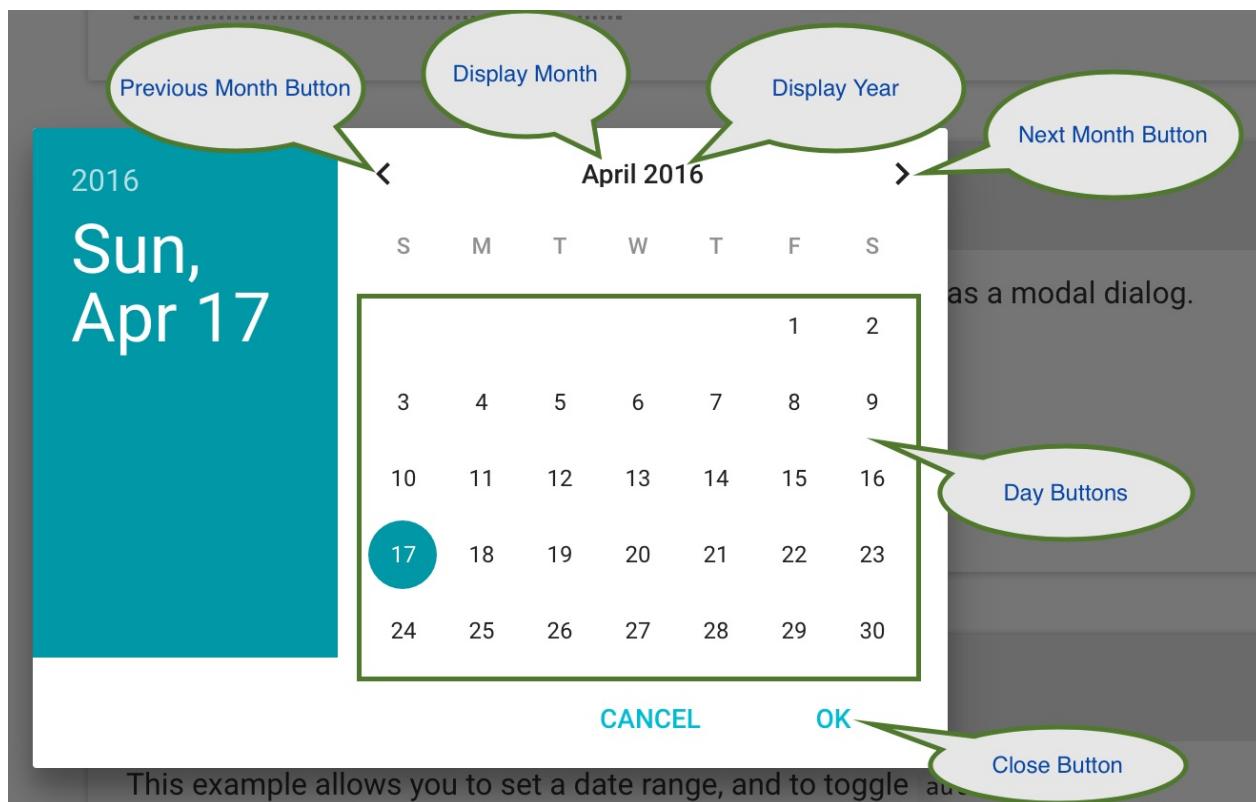


Figure 4. Elements of a Datepicker

In the figure shown above, all annotated elements are the important elements you use to automate datepicker, so you need to use Web Developer Tool to inspect them to find out the HTML codes behind them. We are going to use these names to refer to the elements in the code. Once you understand the elements of the datepicker, you can write locating and manipulating code to automate the operation of picking a date.

There are many styles of the calendar implementation from popular JavaScript frameworks. Even they have different kinds of look and feel, they all have some common elements. Let us have a look of some popular datepickers.

Some popular datepickers

Here are some popular datepickers, they are developed by different people so the look and feels are different. But, they have some common elements.

Let us take a look of jQuery calendar.

jQuery

JQuery calendar is almost same as Material-UI with different style.

jQuery UI Datepicker



Figure 5. jQuery Datepicker

The major difference is JQuery calendar we configured doesn't have Cancel and OK buttons.

Next let us look at the calendar provided for ReactJS.

ReactJS

ReactJS doesn't have built-in datepicker but there are many datepickers written for ReactJS, the one we choose as the example is from Hackerone and it has the same elements as jQuery.



Figure 6. ReactJS Datepicker

There is still difference but the difference can only be seen by Web Developer Tool. We will take about it when automating ReactJS datepicker.

And next is Bootstrap.

Bootstrap

Bootstrap also has the same elements as jQuery.

Bootstrap Datepicker

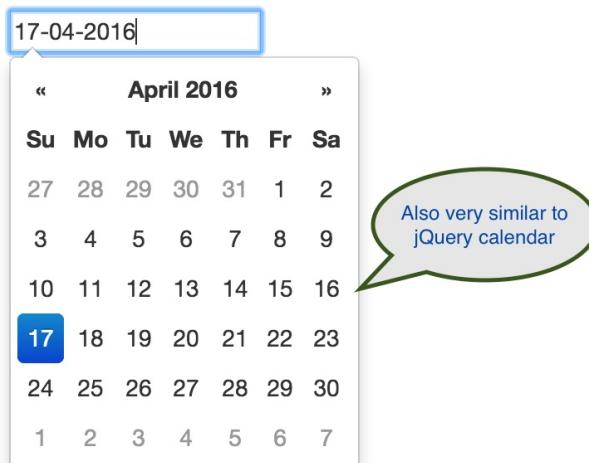


Figure 7. Bootstrap Datepicker

Also we need to find out the real difference by using Web Devoloper Tool.

And one of the oldest datepicker, JsDatePick.

JsDatePick

Unlike the other datepickers, JsDatePick has Previous and Next Year buttons so it makes it quicker to pick a date far from today. Also the week starts on Monday.

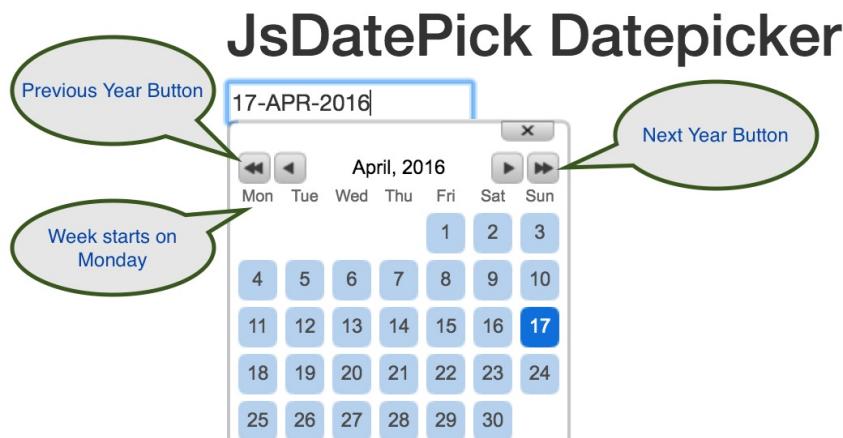


Figure 8. JsDatePick Datepicker

How about Polymer?

Polymer

Polymer also doesn't have its own datepicker. There are many datepickers used with it. One is built on top of Material-UI Date Picker. And there are other choices available. So we don't have example for Polymer. If you need to automate datepickers

using Polymer, you can refer to these examples. As soon as the page is rendered on browser, it doesn't matter what libraries are used to build, it is just some HTML elements. So we can use Selenium WebDriver to automate them.

The similarity among these datepickers suggests us that we may be able to use a common codebase to automate these datepickers. But we don't know that yet, and we need to start from automating one datepicker, jQuery datepicker.

Why jQuery

Besides ReactJS, we also thought of using Material-UI Date Picker as the first example since they are the most trendy JavaScript libraries when this book is written. But we have learnt in Chapter 6 that it is difficult to automate ReactJS. Also since Material-UI Date Picker uses **CSS Transition**, we will have other difficulties when automating it. Those details to solve these problems may distract you from learning the principles of developing a framework. So we decide to choose another datepicker as the first example to automate. JQuery is widely used, and its elements are easy to locate and click.

We will show you how to automate Material-UI Date Picker in Chapter 18, as well as all other datepickers introduced in this chapter.

We are going to inspect the elements on jQuery datepicker and find out the locators we can use to locate them.

Choosing locators for the elements of a jQuery datepicker

We will inspect jQuery datepicker the same way we inspect the calendar in Figure [Elements of a Datepicker](#) and find all HTML codes for these elements on calendar, based on the HTML codes, we can decide what locators to use and how to apply the technique introduced in Chapter 12 to use locator supplier enum constants to organize them.

We learnt from earlier chapters that if an `id` attribute is provided for an element, we had better use it since it is the most stable and efficient locator to use. For example, the trigger and calendar elements have `id` associated with them, so we are going to use `By.ById` locator,

Trigger

Trigger has id "datepicker" so we can use it,

```
<input type="text" id="datepicker" class="hasDatepicker">
```

We can use `By.id("datepicker")` to locate Trigger as a `Element`. Then we can use the `click` method to click the trigger element to popup the calendar.

Calendar

Calendar also has id "ui-datepicker-div" for us to use,

```
<div id="ui-datepicker-div"
    class="ui-datepicker ui-widget ui-widget-content
        ui-helper-clearfix ui-corner-all"
    style="position: absolute; top: 31px; left: 37.484375px; z-index: 1;"
    display: "block">
    ...
</div>
```

We can use `By.id("ui-datepicker-div")` to locate Calendar. We locate Calendar so we can search other elements within the Calendar search scope. That is because `id` attribute is not provided for following elements. We need to use other locators to locate them, as we saw in chapter 2, we can narrow down the search by searching within an element. See technique 4 for more details. Also we can only read or click the following elements while the calendar is displayed.

Display Month

Display month has a class `"ui-datepicker-month"` and we can use it, but before we can use it to read the text, we need to make the calendar visible first, locate the calendar and then locate it using `By.className("ui-datepicker-month")` locator within the Calendar.

```
<span class="ui-datepicker-month">February</span>
```

Then we can use `getText` method to read the text.

Display Year

We can apply the same locating mechanism to locate Display Year since it has a `class` attribute `"ui-datepicker-year"`.

```
<span class="ui-datepicker-year">2016</span>
```

We can use `By.className("ui-datepicker-year")` to locate Display Year within the Calendar and read its text using `getText`.

We need to use logic to find out whether we need to click Previous or Next Month Button and how many times we need to click it in order to flip the calendar to the month we are picking.

Previous Month Button

We need to use Previous Month Button if the date to pick is in the past. It has a `class` attribute `"ui-datepicker-prev"`.

```
<a class="ui-datepicker-prev ui-corner-all" data-handler="prev"
  data-event="click" title="Prev">
  <span class="ui-icon ui-icon-circle-triangle-w">Prev</span>
</a>
```

We can use `By.className("ui-datepicker-prev")` to locate Previous Month Button within the Calendar.

Next Month Button

We need to use Next Month Button if the date to pick is in the future. It has a `class` attribute `"ui-datepicker-next"`.

```
<a class="ui-datepicker-next ui-corner-all" data-handler="next"
  data-event="click" title="Next">
  <span class="ui-icon ui-icon-circle-triangle-e">Next</span>
</a>
```

We can use `By.className("ui-datepicker-next")` to locate Next Month Button within the Calendar.

Day buttons

Once we reach the target month, we need to click the day on the monthly calendar to choose the day. For jQuery datepicker, it is a link so we can use `By.linkText("18")` to pick 18th. But it may not be a link on other calendars. So we need to use other locating methods.

```
<a class="ui-state-default" href="#">18</a>
```

We learnt from earlier chapters that enum is a good practice to organize locators. And we are going to define enum constants to organize these locators, as illustrated by following table.

Table 2. Elements, locators and enums for jQuery datepicker

Elements	Locators	Enum constants
Trigger	<code>By.id("datepicker")</code>	<code>JQueryById.TRIGGER_BY</code>
Calendar	<code>By.id("ui-datepicker-div")</code>	<code>JQueryByIdCALENDAR</code>
DisplayMonth	<code>By.className("ui-datepicker-month")</code>	<code>JQueryByClassName.MONTH</code>
DisplayYear	<code>By.className("ui-datepicker-year")</code>	<code>JQueryByClassName.YEAR</code>
Previous Month Button	<code>By.className("ui-datepicker-prev")</code>	<code>JQueryByClassName.NEXT_MONTH_BUTTON</code>
Next Month Button	<code>By.className("ui-datepicker-next")</code>	<code>JQueryByClassName.NEXT_MONTH_BUTTON</code>
Day Buttons	<code>By.linkText("18")</code>	N/A

In the table, each row is an element, its locator and the enum constant to wrap the locator. Except the Day buttons, since we don't want to define 31 enum constants for those days.

Now that we have all the locators, we can apply the technique we learnt from Expose locator enums as a `Supplier<By>` interface. Now we create two enum constants.

Locator supplier enum by Id

We create `JQueryById` enum to organize the locators using `By.ById` locators, those locators build from `By.id(id)`, and we can keep adding new locators of `By.ById` to this enum when the project grows,

JQueryById.java

```
CALENDAR("ui-datepicker-div"), (1)  
TRIGGER_BY("datepicker"); (2)
```

1. Used to locate calendar and it will be used in `await` method
2. Used to locate trigger and it will be used in `click` method

Then we can use these enum constants to,

1. Locate the trigger and click it.
2. Locate the calendar popup.

And a locator supplier enum to organize those `By.ByClassName`

Locator suppliers by ClassName

We use this enum to organize `By.ByClassName` locators. Similarly to `JQueryById`, we can keep adding new locators of `By.ByClassName` type to this enum,

JQueryByClassName.java

```
NEXT_MONTH_BUTTON("ui-datepicker-next"), (1)  
NEXT_MONTH_BUTTON("ui-datepicker-prev"), (2)  
MONTH("ui-datepicker-month"), (3)  
YEAR("ui-datepicker-year"); (4)
```

1. Used to locate next month button and it will be used in `click` method
2. Used to locate previous month button and it will be used in `click` method
3. Used to locate display month and it will be used in `getText` method
4. Used to locate display year and it will be used in `getText` method

Then we can use these enum constants to

1. Locate the elements on the calendar popup.
2. Read the text of display year and month.

3. Click the buttons to flip calendar.

4. On the target month, pick a day.

We are introducing the following techniques to accomplish the above tasks.

Implementing the jQuery datepicker class

We are going to implement a datepicker class to automate jQuery datepicker, and gradually refactor it into a general purpose framework, so we can use the framework to automate those datepickers built from other JavaScript libraries.

Choose Parameters Carefully to Simplify API Call

We are going to introduce a technique associated with API design to simplify the method invocation. For example, we don't want the user of the method call to go a long way to think how to construct a `java.util.Date` , `java.util.Calendar` or a Java 8 `java.time.LocalDate` in order to use our API, the API should be as simple as `1, 2, 3` . For a datepicker, its functionality is to assist you to pick year, month and then day. So the most straightforward interface is `Datepicker.pick(year, month, day)` .

In Java language, there are many classes represent the concept of date. They are `java.util.Date` , `java.util.Calendar` or a Java 8 `java.time.LocalDate` . If `pickDate` method took any of them as parameter, we need to construct an object of these classes first. But in the code handling date picking, we need to extract the year, month and day information from this object in order to operate on the jQuery calendar widget. There are conversions back and forth between normal string and the date type object.

Good API should be easy to use and read fluently. To save the trouble of writing above mentioned conversion code, an alternative API for datepicker should be the straightforward one, but we change the sequence of the parameters to make it read more like English language, `pick(Month month, int day, int year)` , When you call this method from the automation code, it is as simple as,

Method call to pick date

```
datepicker.pick(APRIL, 1, 2014) (1)
```

1. This is similar to `LocalDate.of(2014, APRIL, 1)` method except the sequence of the parameters

`APRIL` is an enum constant defined in `java.time.Month`, this code reads more fluent and it is much easier to understand. Here is the method body,

JQueryDatepicker.java

```
public void pick(Month month, int day, int year) {
    LocalDate.of(year, month, day);          (1)
    show();                                (2)
    pickYear(year);                        (3)
    pickMonth(month.ordinal());           (4)
    pickDay(day);                          (5)
}
```

1. Use `java.time.LocalDate` to validate the parameter, `LocalDate` is from Java 8
2. Display the calendar, we are going to move this method into `JQueryCalendar` class
3. Flip the calendar into target year, we are going to move it into `JQueryYearControl` class
4. Flip the calendar into target month, we are going to move it into `JQueryMonthControl` class
5. Click the day item and close the calendar, we are going to move it into `JQueryDayPicker` class

From the code above, we can clearly see not only the client code is simple, but the internal logic is simple as well, we can just specify it in the parameters so they can be used directly.

LocalDate

`LocalDate` originated from Joda Date library and after being used many years by the industry as a better date framework, it is officially incorporated into Java 8.

To create a `LocalDate` object, simply call the factory method `of(year, month, day)`

```
LocalDate thatDay = LocalDate.of(2014, APRIL, 1)
```

The `pick` method of our datepicker class, follows this style.

Even we didn't design the method to take a `LocalDate` as the parameter, but inside the method body, the first statement is to construct a `LocalDate` using the 3 parameters, if they are invalid, the method will throw an exception.

```
datepicker.pick(Month.APRIL, 31, 2012);
```

When we run the above code, it will throw the following exception,

```
java.time.DateTimeException: Invalid date 'APRIL 31'
```

It tells you that the date you try to pick doesn't exist. It is better than telling you an `NoSuchElementException`.

Instead of using extra code to convert a string into an object of `java.util.Date`, `java.util.Calendar` or a Java 8 `java.time.LocalDate`, or introducing a dependency to third party date processing API such as Joda Date library, the new API just takes a `java.time.Month`, two `int` values as parameters. Even generally speaking, the less parameters a method takes, the better. But for this case, it is better to use these 3 parameters. You use type `java.time.Month` to restrict the first parameter to be an enum constant of a `Month` which is type safe. And it is not difficult to guess the second parameter is for day and third one for year. You still need to be careful not to pass negative value as day and year. But we have validation logic to verify the parameters are valid.

Implementing a single datepicker class

We don't use too many methods in this chapter, the most frequently used methods are `until`, `await`, `click`, `getText`, `getUpperText` and `optionalElement`. Here is the class diagram showing where these 6 methods coming from.

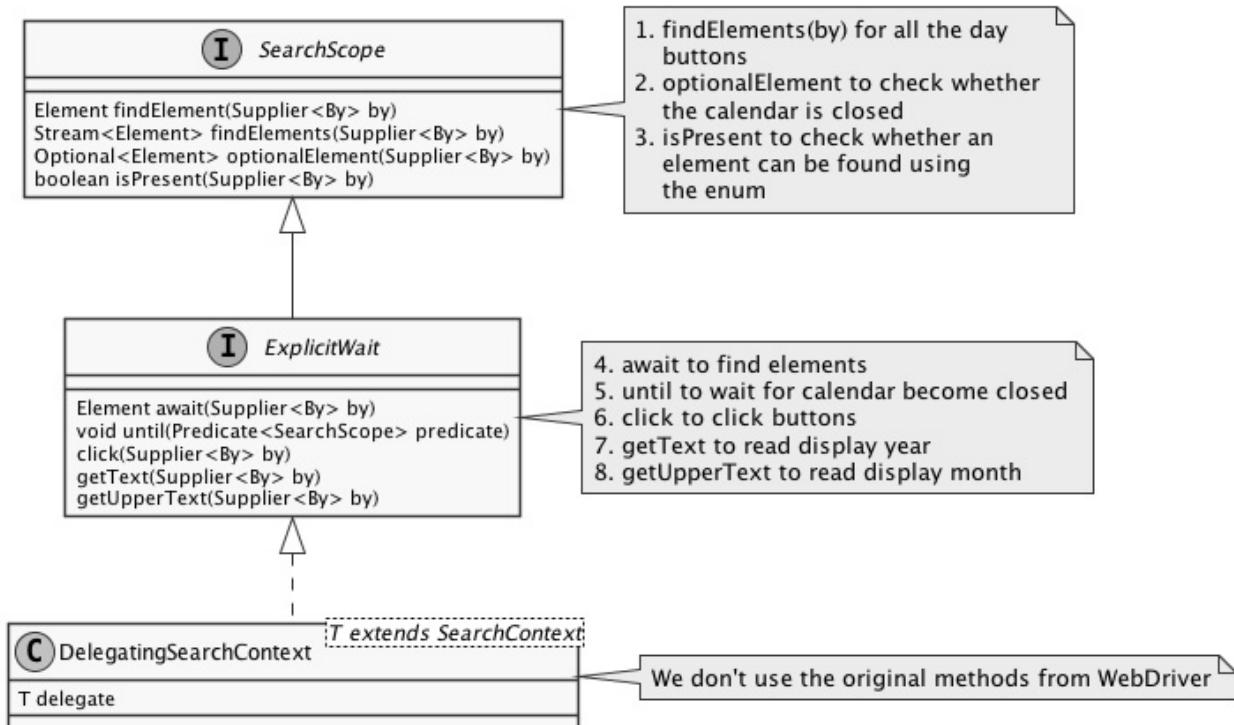


Figure 9. Some frequently used methods

We don't have `Browser` and `Element` in the class diagram, because those methods are not defined in `Browser` and `Element`. `Browser` is a grandson level sub-class of `DelegatingSearchContext` so it has the genes from it and you can call these 6 methods from `Browser` class. Same is `Element` class.

Now we can start to design the class to handle all the date picking logic. First we have the `pick` method from Listing [JQueryDatepicker.java](#), and methods related to display the calendar which is private since it is in the same class,

JQueryDatepicker.java

```

private void show() {
    browser.click(TRIGGER_BY);          (1)
}

```

1. Click the input field to display the calendar

And here are the methods used to pick year. They are in the same class and these methods are declared `private`,

JQueryDatepicker.java

```

private void previousYear() {          (5)
    for (int i = 0; i < 12; i++) {
        previousMonth();
    }
}

private void nextYear() {             (6)
    for (int i = 0; i < 12; i++) {
        nextMonth();
    }
}

private int displayYear() {
    String text = browser.await(CALENDAR).getText(YEAR);  (8)
    return Integer.parseInt(text);   (7)
}

private void pickYear(int year) {     (1)
    if (displayYear() < year) {      (2)
        while (displayYear() != year) { (3)
            nextYear();
        }
    } else if (displayYear() > year) { (4)
        while (displayYear() != year) {
            previousYear();
        }
    }
}

private void previousMonth() {         (9)
    browser.await(CALENDAR).click(PREV_MONTH_BUTTON);
}

private void nextMonth() {           (10)
    browser.await(CALENDAR).click(NEXT_MONTH_BUTTON);
}

```

1. All of the methods are private for now since they are in the same class
2. Determine whether the year to pick is in the past or future
3. Year to pick is in the future in this block
4. Year to pick is in the past in this block so it calls `previousYear()`
5. Since there is no previous year button on the calendar, we just click previous month button 12 times

6. Similar to previousYear except calling `nextMonth()`
7. Use the `Integer.parseInt` to convert the year string into an integer value
8. Read the display year string from the calendar
9. Locate previous month button and click it
10. Locate next month button and click it please see sequence diagram

We can use `nextMonth` as example, to understand the interaction between `JQueryDatepicker`, `Browser` and `Element` classes.

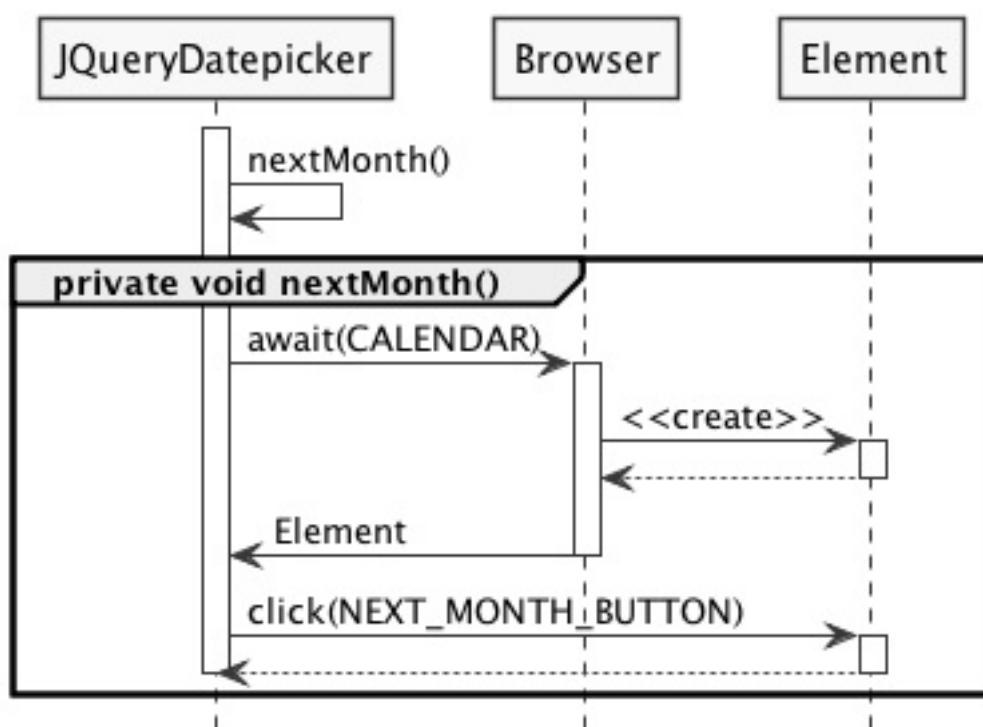


Figure 10. Sequence Diagram of how `nextMonth` method works

And methods related to pick month, which are also `private`,

[JQueryDatepicker.java](#)

```
private void previousMonth() {...}

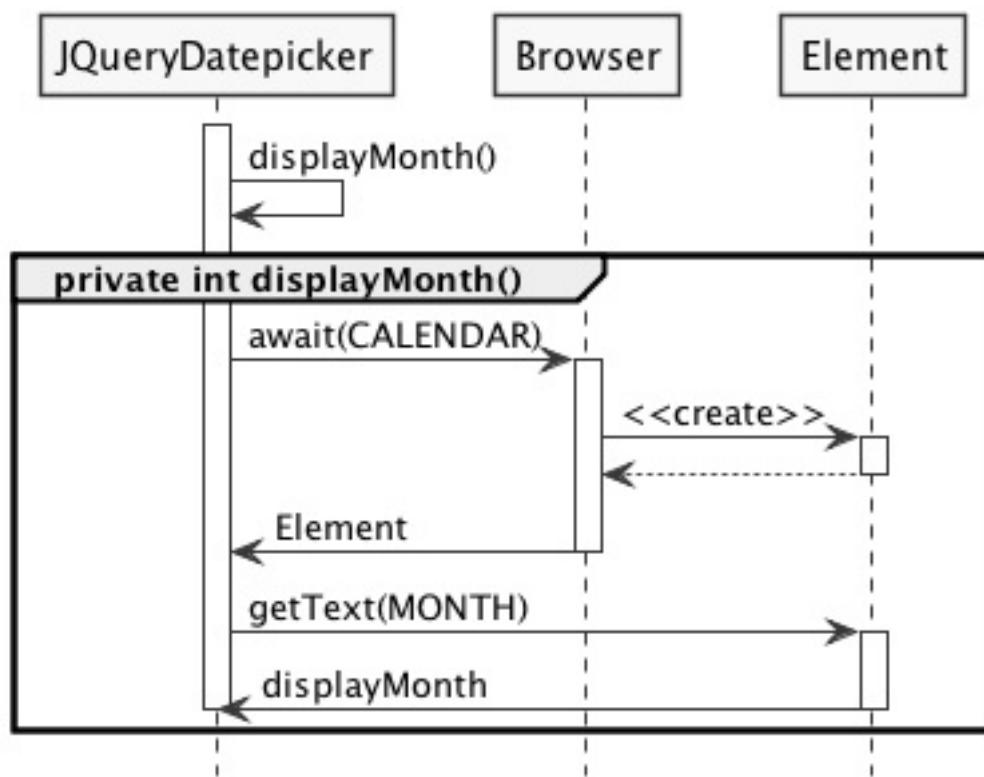
private void nextMonth() {...}

private int displayMonth() {
    String text = browser.await(CALENDAR).getUpperText(MONTH); (2)
    return Month.valueOf(text).ordinal(); (1)
}

private void pickMonth(int month) {
    if (displayMonth() < month) { (3)
        while (displayMonth() != month) { (4)
            nextMonth();
        }
    } else if (displayMonth() > month) {
        while (displayMonth() != month) { (5)
            previousMonth();
        }
    }
}
```

1. Use the `java.time.Month` enum to convert the month and get its ordinal
2. Read the display month from the calendar
3. Determine whether the month to pick is in the past or future
4. Month to pick is in the future
5. Month to pick is in the past so it calls `previousMonth()`

and here is the sequence diagram of `displayMonth` method.

Figure 11. Sequence Diagram of how `displayMonth` method works

As well as the `private` method used to pick the day,

JQueryDatepicker.java

```

private void pickDay(int day) {
    browser.await(CALENDAR)
        .click(new Supplier<By>() { (1)
            @Override
            public By get() {
                return By.linkText(String.valueOf(day));
            }
        }); (2)
    browser.await(new ElementVisible(CALENDAR).negate()); (3)
}
  
```

1. Create an anonymous inner class from interface `Supplier<By>`, this is pre-Java 8 approach and the `Supplier` is from Google Guava library
2. Click the day on the calendar
3. Wait until the calendar is closed

In the method body of `pickDay`, we use `new ElementVisible(CALENDAR).negate()` to wait until the calendar is closed. `ElementVisible` class can be used to check whether an element is visible. The `negate()` method is a method from `Predicate` to reverse the condition so we don't need to have another class `ElementNotVisible`.

ElementVisible.java

```
public class ElementVisible implements Predicate<SearchScope> { (1)

    private final Supplier<By> by; (2)

    public ElementVisible(Supplier<By> by) { (5)
        this.by = by;
    }

    @Override
    public boolean test(SearchScope searchScope) {
        Optional<Element> element = searchScope.optionalElement(by); (3)
        return element.isPresent() && element.get().isDisplayed(); (4)
    }
}
```

1. It implements `Predicate<SearchScope>` interface so it can be used as the parameter for `until` method of `ExplicitWait` interface
2. This is the locator to the element
3. Locate an optional element
4. The logic to check whether it is displayed.
5. From now on we will omit this kind of constructor used to inject the instance variables by assignment only

You can see the code, `JQueryDatepicker` is very clean. For normal automation code, this class is already clean enough to be considered production quality. But it is not good enough to be used as framework code to automate other datepickers. If you pay attention, you will notice, we split those methods into 4 blocks in order to explain the functionalities of those methods. It means the original design is not cohesive and we put unrelated methods inside one class and give it too many responsibilities. We are going to continue refactoring it and move its methods into some calendar control classes to make `JQueryDatepicker` class less crowded.

Improve the performance of calendar flipping

The methods `pickYear` in `JQueryDatepicker` need to read the display value each time after clicking the previous or next month button to determine whether it needs to click again. So each time, it needs to call `displayYear` multiple times in the while loop and it in turn calls the following code through `WebDriver` API,

```
browser.await(CALENDAR).getText(YEAR);
```

This approach automates one situation, somebody watches the calendar while clicking and stops clicking when the display year and display month are the same as the year and month to pick. To improve the performance, the code to flip calendar has been changed to the following. It simulates another person read the current month and year and calculates how many times he need to click the button to go to target month. It may be difficult for a human to calculate the difference but it is a thing a computer is good at.

[JQueryYearPicker.java](#)

```
public void pickYear(int year) {
    int difference = displayYear() - year;      (1)
    if (difference < 0) {                      (2)
        for (int i = difference; i < 0; i++) {
            nextYear();
        }
    } else if (difference > 0) {                (3)
        for (int i = 0; i < difference; i++) {
            previousYear();
        }
    }      (4)
}
```

1. Read display year and calculate the difference between year to pick
2. If the difference is negative, execute the nextYear() method the absolute value of difference times
3. If the difference is positive, execute the previousYear() method the value of difference times
4. In this method, displayYear() only executes once

Same, we can change the logic to pick month as following so the read from display month is also executed once,

[JQueryMonthPicker.java](#)

```

public void pickMonth(int month) {
    int difference = displayMonth() - month; (1)
    if (difference < 0) { (2)
        for (int i = difference; i < 0; i++) {
            nextMonth();
        }
    } else if (difference > 0) { (3)
        for (int i = 0; i < difference; i++) {
            previousMonth();
        }
    } (4)
}

```

1. Calculate the difference between the display month and month to pick
2. If the difference is negative, click the next month button the absolute value of difference times
3. If the difference is positive, click the previous month button the value of difference times
4. In this method, `displayMonth()` only executes once

In the changed `pickMonth` method, `displayMonth` method is only called once. It reduces the times it needs to read data through `WebDriver`.

Automation

From these two way of flipping the calendar, it is analogous to two people clicking the datepicker, one watches the calendar changing while clicking the button and the other calculates how many times he needs to click and counts down the number of clicks with eye closed. Test automation is just to use programming languages to simulate human interaction on web applications. As to how we interact, it still can be different so the logic will be different too.

Introducing delegate classes to split responsibilities

If we want to reuse some date picking logic to automate the datepicker built using other JavaScript framework, we need to extract common logic into some framework classes.

Creating `JQueryCalendar` class to show the calendar

First, we create a `JQueryCalendar` class move `show()` method over and change it to `public`, then we inject this class into `JQueryDatepicker` to be in charge the `show` method that triggers the display of calendar.

`JQueryCalendar.java`

```
public class JQueryCalendar {

    private final Browser browser;

    public void show() {           (1)
        browser.click(TRIGGER_BY);
    }
}
```

1. The method access level changed from `private` to `public`

Next we are going to create a class to pick year.

Creating `JQueryYearPicker` and `JQueryMonthPicker` to pick year and month

And create `JQueryYearPicker` class and move all methods in Listing [JQueryDatepicker.java](#) over, but replace `pickYear` method by improved Listing [JQueryYearPicker.java](#). After these changes, all year picking related methods are in this class,

`JQueryYearPicker.java`

```
public class JQueryYearPicker {

    private final Browser browser;

    public void pickYear(int year) {...} (1)
    ... (2)
}
```

1. This method changed from private to public since it is in another class now
2. These omitted methods are exactly same as the ones in `JQueryYearPicker`

Then another class to pick month.

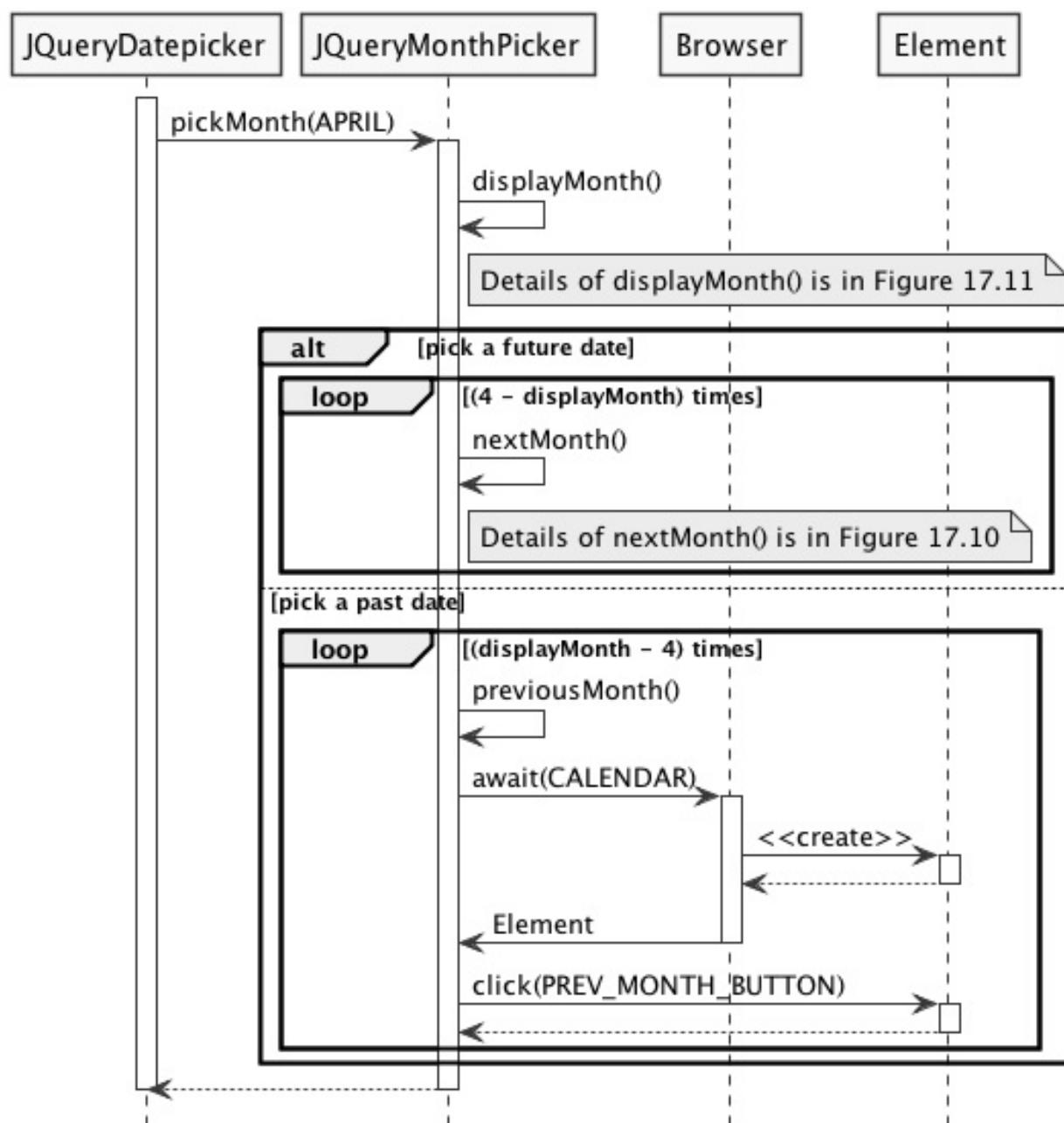
And methods in Listing [JQueryDatePicker.java](#) are moved into `JQueryMonthPicker` and `pickMonth` method is replaced by Listing [JQueryMonthPicker.java](#) so all month related methods are in this class,

[JQueryMonthPicker.java](#)

```
public class JQueryMonthPicker {  
  
    private final Browser browser;  
  
    public void pickMonth(int month) {...} (1)  
    ... (2)  
}
```

1. This method changed from private to public since it is in another class now
2. These omitted methods are exactly same as the ones in `JQueryYearPicker`

Here is the sequence diagram to illustrate the method invocation to pick month.

Figure 12. Sequence Diagram of how `JQueryMonthPicker` works

Due to lacking of Previous Year Button, `JQueryYearPicker` needs to click Previous Month Button 12 times in `previousYear` method. These two methods `previousMonth` and `nextMonth` repeat in both `JQueryYearPicker` and `JQueryMonthPicker` classes. We can remove the duplicates by creating a `JQueryMonthPicker` object during the construction of `JQueryYearPicker` object and use it as a collaborator, as shown in the following code,

`JQueryYearPicker.java`

```

public JQueryYearPicker(Browser browser) {
    this.browser = browser;
    this.monthPicker = new JQueryMonthPicker(browser); (1)
}

```

1. It creates a `JQueryMonthPicker` object and use it to click the month buttons

And modify `previousYear` and `nextYear` methods to use the collaborator.

`JQueryYearPicker.java`

```
private void previousYear() {
    for (int i = 0; i < 12; i++) {
        monthPicker.previousMonth();    (1)
    }
}

private void nextYear() {
    for (int i = 0; i < 12; i++) {
        monthPicker.nextMonth();      (2)
    }
}
```

1. Invoke the `previousMonth` method of the collaborator

2. Invoke the `nextMonth` method of the collaborator

Then delete `previousMonth` and `nextMonth` methods from `JQueryYearPicker` from Listing `JQueryYearPicker.java`. The other methods remain unchanged.

Here is the sequence diagram how to select a future date.

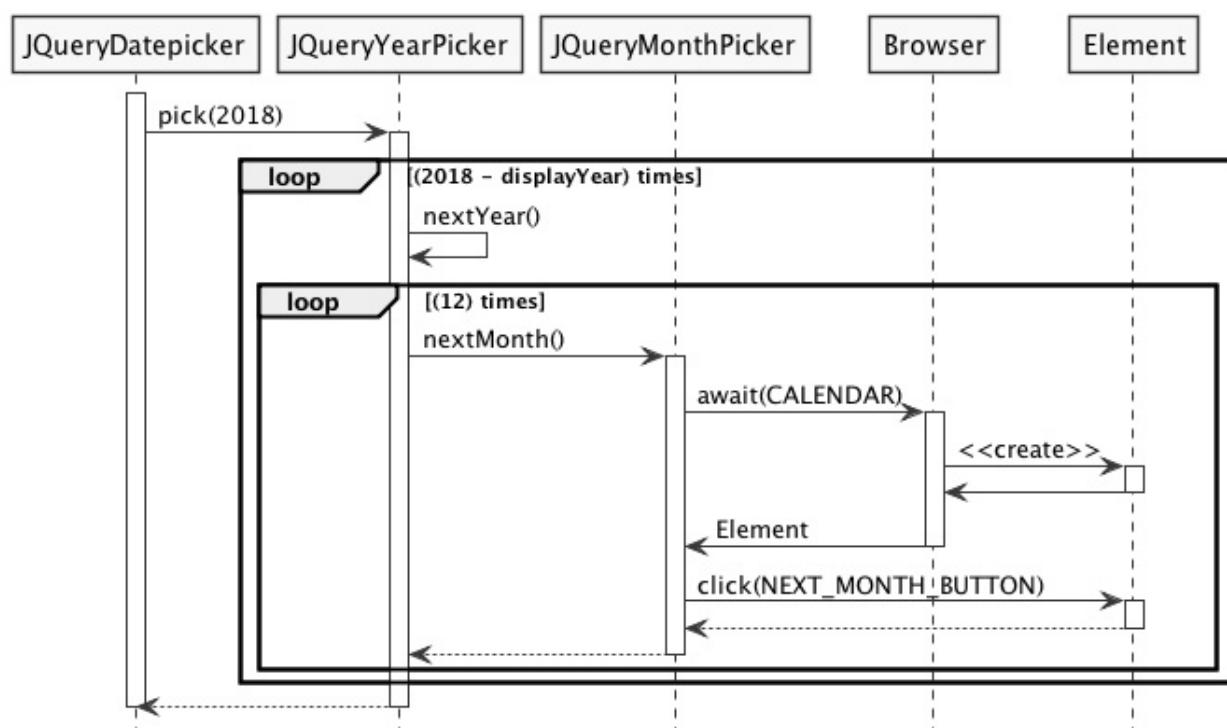


Figure 13. Sequence Diagram of `JQueryYearPicker`

Next, We implement day clicking logic in `JQueryDayPicker` class as method `pickDay`.

Creating `JQueryDayPicker` class to click the day button

The method is same as in Listing [JQueryDatepicker.java](#), we need to change the method from `private` to `public` since it is in `JQueryDayPicker` class.

[JQueryDayPicker.java](#)

```
public class JQueryDayPicker {

    private final Browser browser;

    public void pickDay(int day) {           (1)
        browser.await(CALENDAR)           (2)
            .click(() -> linkText(String.valueOf(day))); (3)
        browser.await(new ElementVisible(CALENDAR).negate());
    }
}
```

1. This method is `public` now
2. This logic doesn't change
3. We use lambda expression replacing the anonymous inner class from now on

Now let us come back to the new `JQueryDatepicker` class.

Inject all collaborators into `JQueryDatepicker`

Now we inject all the collaborators into `JQueryDatepicker` through its constructor so it changed to,

[JQueryDatepicker.java](#)

```

public class JQueryDatepicker {

    private final JQueryCalendar calendar;
    private final JQueryYearPicker yearPicker;
    private final JQueryMonthPicker monthPicker;
    private final JQueryDayPicker dayPicker;

    public void pick(Month month, int day, int year) {
        LocalDate.of(year, month.ordinal(), day);
        calendar.show();                                (1)
        yearPicker.pickYear(year);                      (2)
        monthPicker.pickMonth(month.ordinal());          (3)
        dayPicker.pickDay(day);                         (4)
    }
}

```

1. Call the show method of JQueryCalendar class
2. Call the pickYear method of JQueryYearPicker class
3. Call the pickMonth method of JQueryMonthPicker class
4. Call the pickDay method of JQueryDayPicker class

Here is the sequence diagram of `JQueryDatepicker`,

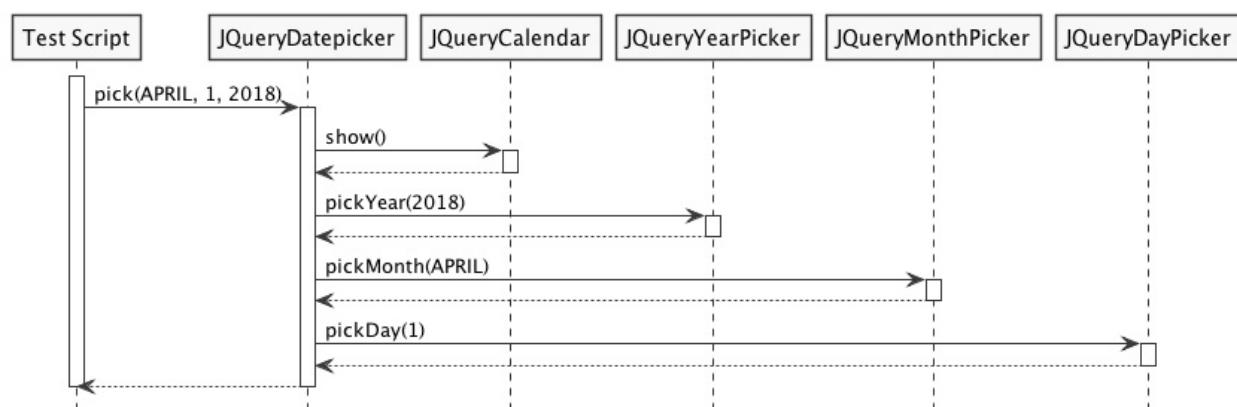


Figure 14. Sequence Diagram of `JQueryDatepicker`

And the class diagram of `JQueryDatepicker`,

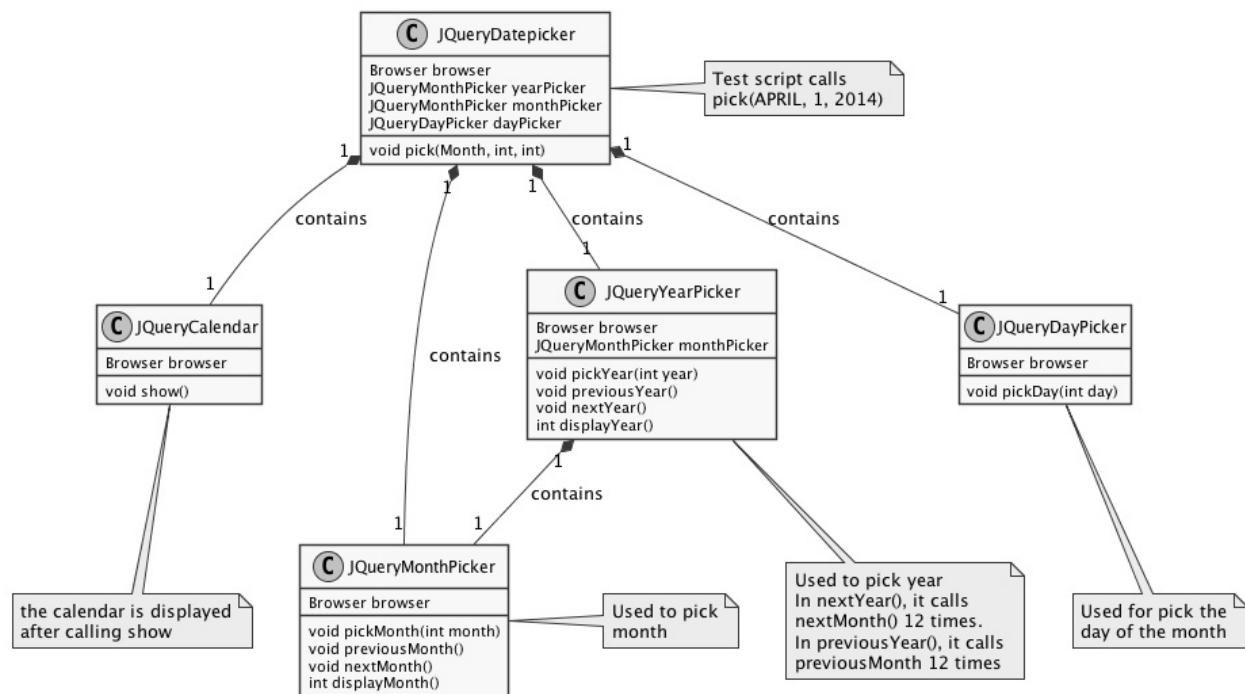


Figure 15. Class Diagram of JQueryDatepicker

The structure is much better now but we will continue to refactor it in the next chapter to remove jQuery specific logic out of the main datepicker class so it can be used as framework code.

As a rule of thumb, delegate pattern often works together with **Dependency Injection** [1] pattern, so you can inject different implementation for the same framework foundation. In the next chapter, we are going to change it to take `Calendar` and `CalendarPicker` classes to operate different datepicker elements.

You can run this test on your computer to watch the calendar flipping and "04/01/2014" is on the input field after the calendar closed.

[JQueryDatePicker_v2_IT.java](#)

```

@Test
public void pickADate() {
    JQueryDatePickerPage.pick(APRIL, 1, 2014);
    assertEquals("04/01/2014", JQueryDatePickerPage.getDate());
}
  
```

When other people read the test, it is very clear what it does, to automate a page to pick April 1st, 2014. This style is very close to English language.

Summary

- You started automating a datepicker by inspecting its elements using Web Developer Tool
- You developed `JQueryDatepicker` class to automate jQuery datepicker.
- You refactored the `JQueryDatepicker` class into a controller class and moved the detailed logic to couple of collaborators and paved the way for a framework.

In the next chapter, we are going to refactor and move jQuery specific implementation out of the `Datepicker` class, which is renamed from `JQueryDatepicker` and inject all collaborators when constructing the `Datepicker` object during runtime. And then we are going to inject other implementations into the same `Datepicker` to automate the datepicker built using other libraries such as Bootstrap, JsDatePicker, ReactJS and Material-UI.

1. http://en.wikipedia.org/wiki/Dependency_injection

Chapter 18: Datepicker Framework

This chapter covers

- Extracting general purpose datepicker framework
- Automating other datepickers using the framework

In this chapter, we will continue from where we left on last chapter, refactor the code into a Datepicker framework and apply the framework in automating datepickers built on other major JavaScript libraries such as Bootstrap, ReactJS, Material-UI and JsDatePick.

By the end of this chapter, not only you will learn the technique and apply it in your day to day work, you can also learn skills to apply the framework to many datepickers, as well as the principles and practices of developing an emergent framework and apply them in your career do develop frameworks to automate other web widgets that are not covered by this book.

Extracting general purpose datepicker framework

In this section, we are going to show you a technique to introduce a framework so we don't need to repeat some classes when automating datepickers built using other JavaScript frameworks.

Introducing a framework for multiple JavaScript implementations

Besides datepicker built using JQuery, there are datepickers built using other JavaScript frameworks, it would be duplicate code if we just copy `JQueryDatepicker` into another class.

The previously demonstrated code are well formed and very concise, but there is jQuery specific code in the classes so they are not ready to be used to automate the datepickers built using other JavaScript frameworks.

We are going to move jQuery specific implementation out of main datepicker classes, so those classes can be used as framework code.

We will rename Listing 17.23 [JQueryDatepicker.java](#) to `Datepicker` to remove all jQuery specific implementation to accept general `Calendar`, `YearPicker`, `MonthPicker` and `DayPicker` classes as parameters.

[Datepicker.java](#)

```
public class Datepicker {

    private final Calendar calendar;          (1)
    private final YearPicker yearPicker;       (2)
    private final MonthPicker monthPicker;     (3)
    private final DayPicker dayPicker;         (4)

    public void pick(Month month, int day, int year) {...} (5)
}
```

1. This variable changed from JQueryCalendar to Calendar
2. This variable changed from JQueryYearPicker to YearPicker
3. This variable changed from JQueryMonthPicker to MonthPicker
4. This variable changed from JQueryDayPicker to DayPicker
5. The method body doesn't change from Listing 17.23

You can see the `Datepicker` has several variables injected from its constructor, a `Calendar` class to display the calendar, a `YearPicker` class to pick year, a `MonthPicker` class to pick month and a `DayPicker` to click the day of the month.

We extracted a common purpose `Datepicker` class. It will work with most of the single month calendar widget, no matter which library it is built upon because it doesn't have dependency on those calendar widget. We then define how `Calendar`, `YearPicker`, `MonthPicker` and `Datepicker` for the constructor of `Datepicker` class

Adding collaborators for `Datepicker` class

Provide a `Calendar` to display the calendar

In Listing 17.18 [JQueryCalendar.java](#), we can see `JQueryCalendar` class is used to trigger the display of the calendar. In its `show` method, `browser` clicks the trigger locator. But to click the trigger of calendar built using other JavaScript library, the locator may be different. So we need a parameter to pass in an instance of other class, and when `show` method is called, it will call a method of this instance, and in that method, we can specify the logic to click the trigger locator. After some research, we decide to use `Consumer` interface from Java 8, [1]. The reason to use `Consumer` is because its return type is `void` so you don't have to add `return null` at the end of each method that implements this interface.

Pre-Java 8 alternative

If you don't use Java 8, you can simply create the following interface in your codebase, [v2_5/Consumer.java](#)

```
public interface Consumer<T> {
    void accept(T t);
}
```

It doesn't matter whether you use this one or the one from Java 8, the only difference is the import statements, you either import this home made class,

```
import swb.ch18datepicker.jquery.v2_5.Consumer;
```

or import this one from Java 8,

```
import java.util.function.Consumer;
```

and the body part of `Calendar` is same,

[Calendar.java](#)

```
public class Calendar {

    private final Browser browser;
    private final Consumer<Browser> trigger;    (1)

    public void show() { (2)
        trigger.accept(browser);      (3)
    }
}
```

1. The Consumer to trigger the display of the calendar
2. To display the calendar
3. Call the accept method to display the calendar

We can define a `Trigger` class to implement `Consumer<Browser>` interface so it can be used as the parameter for the constructor of the `Calendar` class,

[Trigger.java](#)

```
public class Trigger implements Consumer<Browser> { (1)
    @Override
    public void accept(Browser browser) { (2)
        browser.click(TRIGGER_BY); (3)
    }
}
```

1. Since `Trigger` class implements `Consumer<Browser>` interface, it can be used as the parameter for the constructor of `Calendar` class
2. Its `accept` method takes `Browser` as parameter, so we can invoke the `trigger.accept(browser);` in the `show` method of `Calendar` class
3. In turn, it calls `browser.click(TRIGGER_BY);`

To create an instance of `Calendar`, we need to create an instance of `Trigger` class and pass it as the second parameter to the constructor of `Calendar` class, as shown in the following code snippet,

Create Calendar instance

```
Calendar calendar = new Calendar(browser, new Trigger()); (1)
calendar.show(); (2)
```

1. Create an instance of the calendar
2. Call its `show` method and in term call the `accept` method of the `Trigger` class and call `browser.click(TRIGGER_BY);`

A calendar will be displayed after this call.

Here is the sequence diagram of how to trigger the display of the calendar

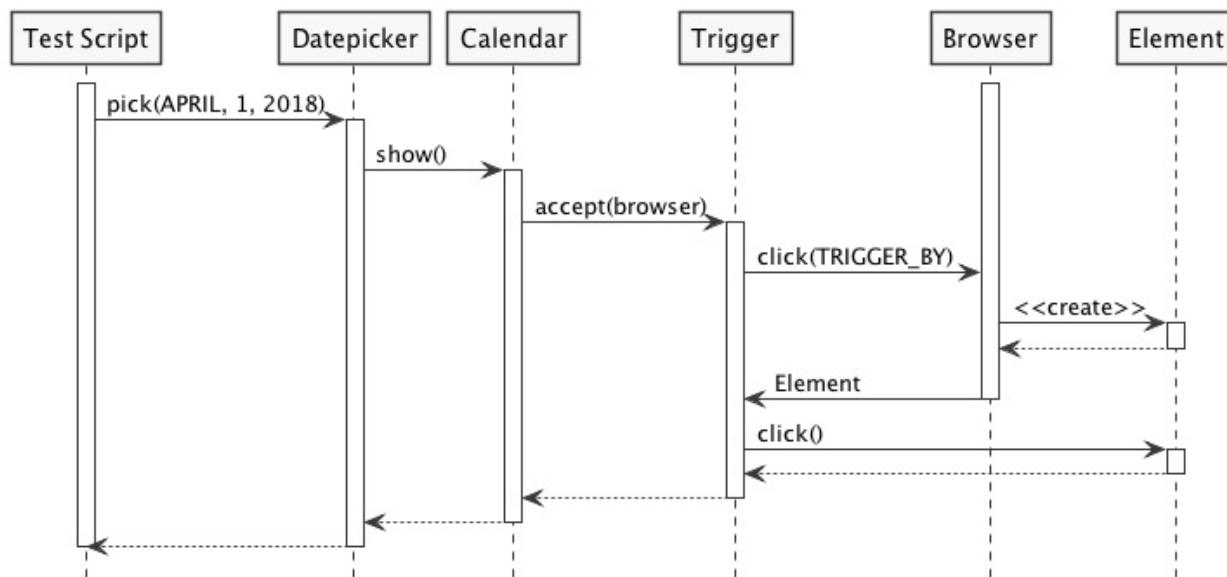


Figure 1. Sequence Diagram of Triggering the Display of Calendar

Consumer vs Function

We did use `Function<Browser, Void>` at the beginning of the writing. At that time, the `Trigger` class was implemented as [v2_5/Trigger.java](#)

```

public class Trigger implements Function<Browser,Void> {

    @Override
    public Void apply(Browser browser) {
        browser.click(TRIGGER_BY);
        return null;
    }
}
  
```

This `Trigger` class implements `Function<Browser, Void>` interface and return `Void` type, so it needs an extra line of `return null` at the end of the method.

`Consumer` interface from Java 8 is better in this case.

Implementing MonthPicker class to pick month

We then define a `MonthPicker` class in charge of flipping the calendar to the desired month. We learnt from Chapter 17 that `MonthPicker` class need to be able to read "Display Month" from calendar and click "Previous Month Button" or "Next Month Button", depending on the direction of the month it wants to pick. In Listing 17.20, those responsibility is handled by the methods inside `MonthPicker` class. We are going to treat the buttons the same way we treat the trigger, to let them be a

`Consumer<Browser>` interface and implement the logic inside `accept` method to click the button. Since we need to read the "Display Month" as an integer value, we define that as an `Function<Browser, Integer>`.

These are the instance variables of `MonthPicker` class, they are passed by the constructor of the class.

MonthPicker.java

```
private final Browser browser;                                (1)
private final Consumer<Browser> previousMonth;           (2)
private final Consumer<Browser> nextMonth;                 (3)
private final Function<Browser, Integer> displayMonth;     (4)
```

1. A `Browser` which provides access to `WebDriver`
2. A `Consumer<Browser>` to click previous month button
3. A `Consumer<Browser>` to click next month button
4. A `Function<Browser, Integer>` to read the display month on calendar

And the `pick` method has the same logic as Listing 17.20, what has changed is that those original methods of `JQueryMonthPicker` are replaced by the methods from those instance variables `previousMonth`, `nextMonth` and `displayMonth`

Table 1. Relocation of original JQueryDatepicker methods for month related operations

Original methods of JQueryMonthPicker	New methods in MonthPicker	Host Classes of the new methods
<code>displayMonth()</code>	<code>displayMonth .apply(browser)</code>	<code>DisplayMonth</code>
<code>nextMonth()</code>	<code>nextMonth .accept(browser)</code>	<code>NextMonth</code>
<code>previousMonth()</code>	<code>previousMonth .accept(browser)</code>	<code>PreviousMonth</code>

Here is the `pick` method of `MonthPicker`,

MonthPicker.java

```

public void pick(int month) {
    int difference = displayMonth.apply(browser) - month;      (1)
    if (difference < 0) {                                         (2)
        for (int i = difference; i < 0; i++) {
            nextMonth.accept(browser);                            (3)
        }
    } else if (difference > 0) {                                     (4)
        for (int i = 0; i < difference; i++) {                      (5)
            previousMonth.accept(browser);
        }
    }
}

```

1. To calculate how many times it need to click, if `displayMonth` is 8 and month is 11, then the difference is -3
2. If the difference is negative, it means the target is in the future, so need to click next month button
3. Clicking next month button many times, according to the difference between `displayMonth` of month
4. If the difference is positive, it means the target is in the past, so need to click previous month button
5. Clicking previous month button many times, according to the difference between `displayMonth` of month

And we can use the host class names in the column 3 of the table to define the classes to be used as parameters for the constructor of `MonthPicker` class.

`PreviousMonth` class for the second parameter of the constructor of `MonthPicker` class,

PreviousMonth.java

```

public class PreviousMonth implements Consumer<Browser> { (1)
    @Override
    public void accept(Browser browser) {
        browser.await(CALENDAR).click(PREV_MONTH_BUTTON); (2)
    }
}

```

1. It implements `Consumer<Browser>`
2. This is same as `previousMonth` method in `JQueryMonthPicker` class

`NextMonth` class for the third parameter of the constructor of `MonthPicker` class,

NextMonth.java

```
public class NextMonth implements Consumer<Browser> {    (1)
    @Override
    public void accept(Browser browser) {        (2)
        browser.await(CALENDAR).click(NEXT_MONTH_BUTTON);
    }
}
```

1. It implements `Consumer<Browser>`

2. This is same as `nextMonth` method in `JQueryMonthPicker` class

`DisplayMonth` class for the fourth parameter of the constructor of `MonthPicker` class,

DisplayMonth.java

```
public class DisplayMonth implements Function<Browser, Integer> {    (1)
    @Override
    public Integer apply(Browser browser) {        (2)
        String text = browser.await(CALENDAR).getUpperText(MONTH);
        return Month.valueOf(text).ordinal();
    }
}
```

1. It implements `Function<Browser, Integer>`

2. This is same as `displayMonth` method in `JQueryMonthPicker` class

Then we can create an instance of `MonthPicker` class by calling its constructor with the required parameters, `browser`, `new PreviousMonth()`, `new NextMonth()` and `new DisplayMonth()`.

```
new MonthPicker(browser,
    new PreviousMonth(), new NextMonth(), new DisplayMonth());
```

And this instance of `MonthPicker` can be used as a parameter for the constructor of `DatePicker` class.

Changing `MonthPicker` to `CalendarPicker` class to pick both month and year

We are extracting a `YearPicker` class in charge of flipping the calendar to the desired year, but we notice it has exactly same type of variables as `MonthPicker` class. The logic of the `pick` method of `YearPicker` and `MonthPicker` are same except that use different variable names, so we decide to use one class `CalendarPicker`.

CalendarPicker.java

```
public class CalendarPicker {

    private final Browser browser;
    private final Consumer<Browser> previous;      (1)
    private final Consumer<Browser> next;            (2)
    private final Function<Browser, Integer> displayValue; (3)

    void pick(int value) {...} (4)
}
```

1. It can be previous month or previous year
2. It can be next month or next year
3. It can be display month or display year
4. The logic is same as before except the variable names changed accordingly

Same as month picking classes, we need some year picking classes as well,

Table 2. Relocation of original JQueryDatepicker methods for year related operations

Original methods of JQueryYearPicker	New methods in CalendarPicker	Host Classes of new methods
<code>displayYear()</code>	<code>displayValue.apply(browser)</code>	<code>DisplayYear</code>
<code>nextYear()</code>	<code>next.accept(browser)</code>	<code>NextYear</code>
<code>previousYear()</code>	<code>previous.accept(browser())</code>	<code>PreviousYear</code>

And we can provide implementations for jQuery and use them as the parameters for the constructor of the `YearPicker`.

`PreviousYear` class for the second parameter of the constructor of `YearPicker` class,

PreviousYear.java

```
public class PreviousYear implements Consumer<Browser> { (1)

    private final PreviousMonth previousMonth = new PreviousMonth();

    @Override
    public void accept(Browser browser) { (2)
        for (int i = 0; i < 12; i++) { (3)
            previousMonth.accept(browser);
        }
    }
}
```

1. It implements `Consumer<Browser>`
2. This method is the `previousYear` in `JQueryYearPicker` class
3. Same as `JQueryYearPicker` class, it uses `PreviousMonth` to click the previous month button 12 times.

`NextYear` class for the third parameter of the constructor of `YearPicker` class,

NextYear.java

```
public class NextYear implements Consumer<Browser> { (1)

    private final NextMonth nextMonth = new NextMonth();

    @Override
    public void accept(Browser browser) { (2)
        for (int i = 0; i < 12; i++) { (3)
            nextMonth.accept(browser);
        }
    }
}
```

1. It implements `Consumer<Browser>`
2. This method is the `nextYear` in `JQueryYearPicker` class
3. Same as `JQueryYearPicker` class, it uses `NextMonth` to click the next month button 12 times.

`DisplayYear` class for the fourth parameter of the constructor of `YearPicker` class,

DisplayYear.java

```
public class DisplayYear implements Function<Browser, Integer> { (1)
    @Override
    public Integer apply(Browser browser) { (2)
        String text = browser.await(CALENDAR).getText(YEAR);
        return Integer.parseInt(text);
    }
}
```

1. It implements `Function<Browser, Integer>`
2. This method is the `displayYear` in `JQueryYearPicker` class

We can create an instance of `CalendarPicker` class by calling its constructor and pass the required parameters such as `browser` , `new PreviousYear()` , `new NextYear()` and `new DisplayYear()` ,

Create `CalendarPicker` instance to pick year

```
new CalendarPicker(browser, new PreviousYear(), new NextYear(), new DisplayYear()),
```

And this instance of `CalendarPicker` can be used as a parameter for the constructor of `DatePicker` class.

Here is the sequence diagram of how `CalendarPicker` works,

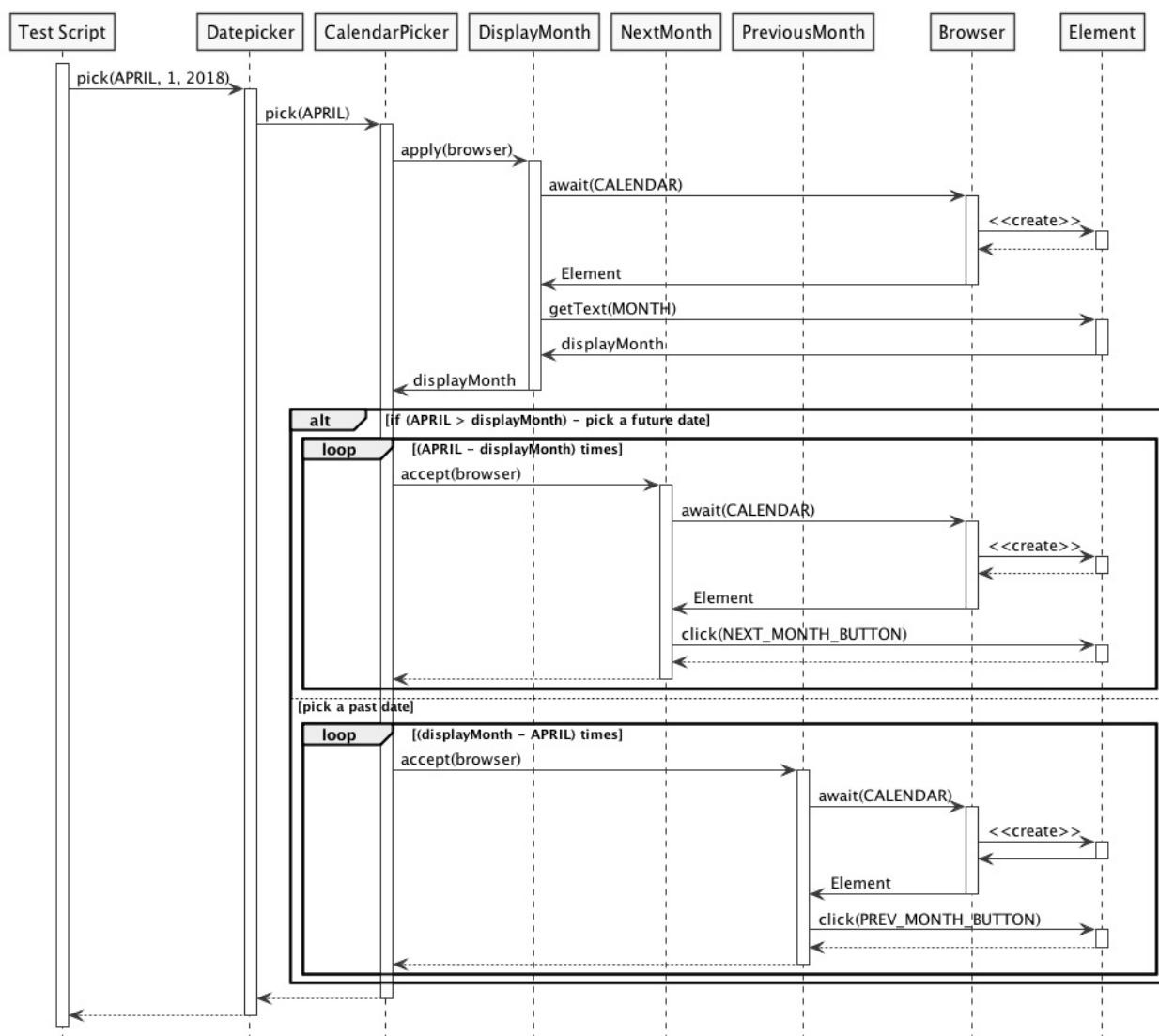


Figure 2. Sequence Diagram of how CalendarPicker works

Providing an interface for picking a day from the month

We add this `DayPicker` interface to pick the day from the calendar month. Since we don't have a way to template the operations to pick a day, so we make it an interface.

[DayPicker.java](#)

```

public interface DayPicker {
    void pick(int day);
}

```

And have the `JQueryDayPicker` implementing this interface and change its `pickDay` method to `pick`, [JQueryDayPicker.java](#)

```

public class JQueryDayPicker implements DayPicker {...}

```

After these changes, `Datepicker` class evolves into the following form, but the body of `pick` method doesn't change from Listing , what has changed is the types of the instance variables.

Datepicker.java

```
private final Calendar calendar;           (1)
private final CalendarPicker yearPicker;   (2)
private final CalendarPicker monthPicker;  (3)
private final DayPicker dayPicker;         (4)
```

1. The Calendar
2. The control to pick the year from the calendar, it changed from YearPicker to CalendarPicker
3. The control to pick the month from the calendar, it changed from MonthPicker to CalendarPicker
4. The control to pick the day from the calendar

Here is the class diagram of current `Datepicker` ,

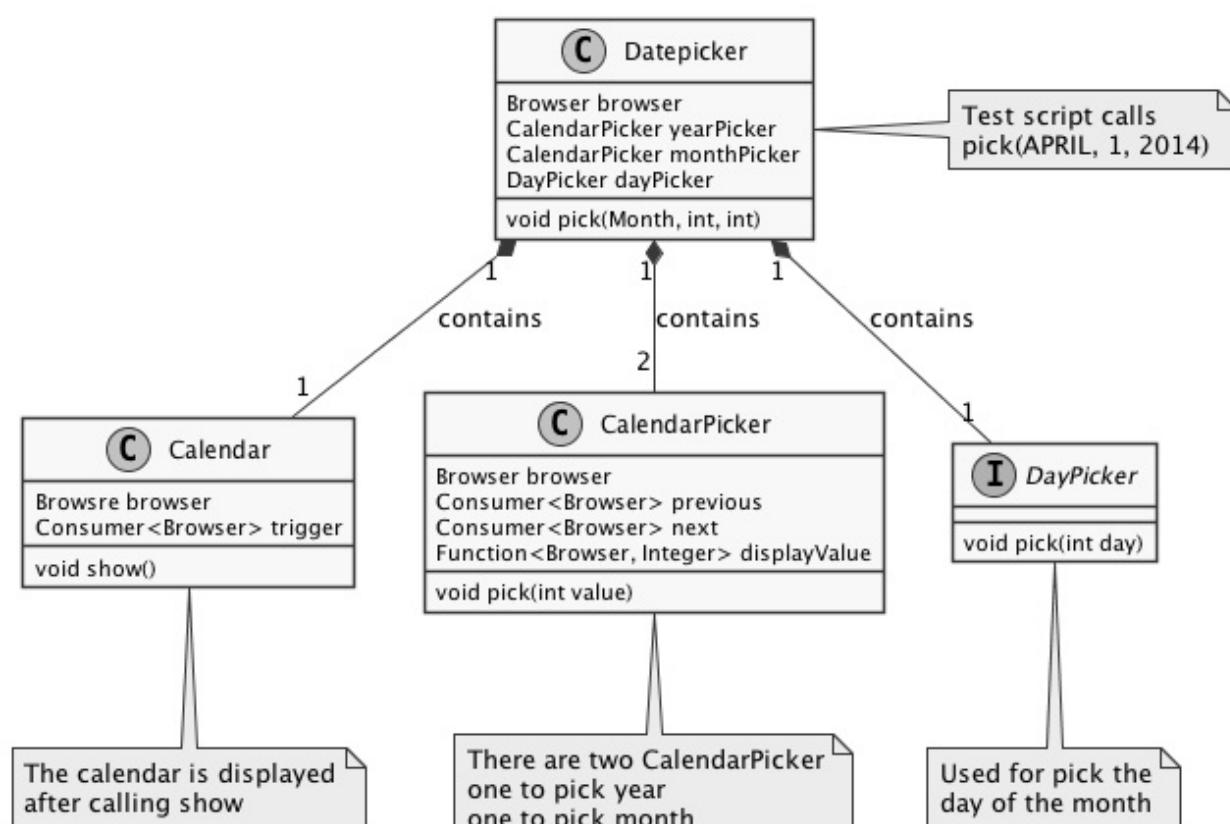


Figure 3. Class Diagram of Datepicker

You can see, it is simpler than the previous version and only have 4 framework classes.

Then We use page object pattern to organize the responsibilities of the page.

Adding a Page Object to create the `Datepicker` object

Applying what we have learnt from Chapter 5, we create `JQueryDatepickerPage` class with the `pick` method to pick the date and `getDate` method to read the date it picks, and the `pick` method just delegates the call to the datepicker instance variable it creates during construction time. You can see how to create an instance of `Datepicker` class in the constructor of `JQueryDatepickerPage` class,

`JQueryDatepickerPage.java`

```
this.datepicker = new Datepicker( (1)
    new Calendar(browser, new Trigger()),
    new CalendarPicker(browser,
        new PreviousYear(), new NextYear(), new DisplayYear()),
    new CalendarPicker(browser,
        new PreviousMonth(), new NextMonth(), new DisplayMonth()),
    new JQueryDayPicker(browser));
```

1. Creating an instance of `Datepicker` and injecting its instance variables using constructor injection

The it can tested by the test `JQueryDatepicker_v3_IT`. As of the writing, it flips the calendar towards the future direction.

While extracting jQuery specific logic out of the framework classes, we have implemented many classes, for example, `Trigger`, `PreviousMonth`, `PreviousYear` and so on to represent the controls on calendar. This approach resulted in too many classes to be managed. One way to remove those classes is to use Lambda Expression from Java 8. When we automate other datepickers, we will replace those classes instantiation with lambda expression.

Automating other datepickers

We then use the same framework to automate the datepicker built using Bootstrap.

Automating Bootstrap datepicker

Bootstrap is a very popular web framework, so we are going to use its datepicker to demonstrate the applicability of this Datepicker framework. We are going to use Web Developer Tool to inspect the elements of this datepicker.

Identifying the elements of Bootstrap datepicker

Most of the elements on Bootstrap calendar are similar to jQuery calendar except they use different `class` attributes. But there is a major difference, unlike jQuery, whose display year and display month are in two separate HTML `span` elements, on Bootstrap calendar, the month and year displayed are on same HTML `th` element.

```
<th colspan="5" class="datepicker-switch">February 2016</th> (1)
```

How to make it work with the framework we developed for jQuery datepicker? We are going to need a class to extract the month and year value from the same display element, but first we need to code to locate those elements.

Once we identify all the important elements on the Bootstrap datepicker, we can define locator supplier enum `BootstrapByClassName` to implement `Supplier<By>`.

[BootstrapByClassName.java](#)

```
CALENDAR("datepicker-days"),          (1)
TRIGGER_BY("trigger"),              (2)
NEXT_MONTH_BUTTON("next"),          (3)
PREV_MONTH_BUTTON("prev"),          (4)
DISPLAY_MONTH_YEAR("datepicker-switch"); (5)
```

1. Locator for the calendar
2. Locator for the calendar trigger
3. Locator for next month button
4. Locator for previous month button
5. Locator for display month and year

Next, we are going to implement some function to click the elements to flip the calendar backwards or forwards.

Implementing Bootstrap specific functions using lambda expression

Unlike jQuery, we implemented many classes for the functions, we are going to use lambda expression to create the `Datepicker` instance. But this works only on Java 8.

BootstrapDatePickerPage.java

```
this.datepicker = new Datepicker(
    new Calendar(browser,          (6)
        b -> browser.click(TRIGGER_BY)  (1)
    ),
    new CalendarPicker(browser,     (7)
        b -> previousYear(),   (2)
        b -> nextYear(),       (3)
        b -> displayYear()     (4)
    ),
    new CalendarPicker(browser,     (8)
        b -> previousMonth(),
        b -> nextMonth(),
        b -> displayMonth()
    ), new BoostrapDayPicker(browser));      (5)
```

1. Create a `Consumer<Browser>` object and when its `accept` method is called, call `broswer.click(TRIGGER_BY)` for the second parameter to `Calendar`
2. Create a `Consumer<Browser>` object and when its `accept` method is called, call `this.previousYear()` for the second parameter to `YearPicker`
3. Create a `Consumer<Browser>` object and when its `accept` method is called, call `this.nextYear()` for the third parameter to `YearPicker`
4. Create a `Function<Browser, Integer>` object and when its `apply` method is called, call `this.displayYear()` for the fourth parameter to `YearPicker`
5. Create a `BoostrapDayPicker` object for the fourth parameter to `Datepicker`
6. Create a `Calendar` object for the first parameter to `Datepicker`
7. Create a `CalendarPicker` object for the second parameter to `Datepicker`
8. Create a `CalendarPicker` object for the second parameter to `Datepicker`

You use Lambda expression to create instance of those classes, so you don't need to define `Trigger`, `PreviousYear`, `NextYear`, `DisplayYear`, `PreviousMonth`, `NextMonth` and `DisplayMonth` classes. But you still need to add those methods inside

`BootstrapDatepickerPage` to provide correspondent functions.

Here are some methods used to pick month and the methods for pick year are quite similar.

BootstrapDatepickerPage.java

```
private int displayMonth() {
    return TO_MONTH.apply(extract(browser, 0)).ordinal();
}

private void previousMonth() {
    browser.await(CALENDAR).click(PREV_MONTH_BUTTON);  (3)
}

private void nextMonth() {
    browser.await(CALENDAR).click(NEXT_MONTH_BUTTON);  (4)
}

private String extract(Browser browser, int i) {  (1)
    return browser.await(CALENDAR)
        .getText(DISPLAY_MONTH_YEAR).split(" ")[i];
}
```

`BootstrapDayPicker` is the day picker for Bootstrap. And Bootstrap calendar behaves slightly different from jQuery calendar, it doesn't close after `WebDriver` clicks the day button, so we need to add logic to click outside the calendar to close it.

BootstrapDayPicker.java

```
public class BootstrapDayPicker implements DayPicker {

    private Browser browser;

    @Override
    public void pick(int day) {
        browser.await(CALENDAR).findElements(TD)          (2)
            .filter(e -> e.getText().equals(String.valueOf(day)))  (3)
            .findFirst()      (5)
            .get()
            .click();           (4)
        browser.click(FORM);                (1)
        browser.await(new ElementVisible(CALENDAR).negate()); (6)
    }
}
```

1. After clicking the day from the month, bootstrap doesn't closed the calendar so need to click outside the calendar to close it
2. Find the calendar and then find all `td` elements on the calendar
3. Find all `td` elements with the text same as the day string
4. Click it, that is the day we want to pick
5. The first `td` same as the day string is the only one we are looking for
6. Wait for the calendar become invisible

You can run the following test to see how it works, and the style of this test is more concise than the previous listing,

[BootstrapDatepicker_v1_IT.java](#)

```
@Test
public void pickADate() {
    new BootstrapDatepickerPage(browser)
    {
        { (1)
            pick(APRIL, 1, 2015); (2)
            assertEquals("01-04-2015", getDate()); (3)
        }
    };
}
```

1. Create an instance of anonymous subclass `BootstrapDatepickerPage` class and the code within the inner bracket is anonymous constructor
2. Call `pick` method of `BootstrapDatepickerPage`
3. Call `getDate` method of `BootstrapDatepickerPage`

Now we can conclude that the framework can be used in the automation for two datepicker implementations.

As of today, ReactJS became more and more popular, can we use this framework to automate the datepicker built upon ReactJS?

Automating ReactJS Datepicker crafted by Hackerone

ReactJS became more and more popular now, so we use this ReactJS Datapicker crafted by Hacherone to demonstrate the applicability of this Datepicker framework. But it is not the only datepicker available for ReactJS.

Identifying the elements of ReactJS Datepicker

Most of the elements on ReactJS calendar are also similar to jQuery and Bootstrap calendars except they use different `class` attributes. Similar as Bootstrap, the month and year displayed are on one `div` element.

```
<div class="react-datepicker__current-month">April 2016</div>
```

So we also need to extract the month and year value from the `div` element, just like what we did for Bootstrap.

We define locator supplier enum `ReactByClassName` to implement `Supplier<By>`,

[ReactByClassName.java](#)

```
TRIGGER_CONTAINER("react-datepicker__input-container"),
TRIGGER_BY("ignore-react-onclickoutside"),
CALENDAR("react-datepicker"), (1)
NEXT_MONTH_BUTTON("react-datepicker__navigation--next"), (2)
PREV_MONTH_BUTTON("react-datepicker__navigation--previous"), (3)
DISPLAY_MONTH_YEAR("react-datepicker__current-month"); (4)
```

1. Locator for the calendar
2. Locator for next month button
3. Locator for previous month button
4. Locator for display month and year

We learnt from Chapter 6 that we can't use its `class` attribute to locate input field. And we are going to use an Xpath alternative locating method to find the trigger,

[ReactByXpath.java](#)

```
TRIGGER_BY("//*[@id='app']/descendant::input"); (1)
```

1. It means find something with id "app" and the first input field on that element

We run this test and it displays the calendar.

[FindByXpath_v2_IT.java](#)

```
@Test  
public void trigger() {  
    browser.click(ReactByXpath.TRIGGER_BY);  
}
```

And you can see how clean the code becomes after using framework.

Next, we are going to implement some function to click the elements to flip the calendar backwards or forwards.

Implementing ReactJS specific functions

Unlike jQuery, we used individual classes for each functions, and unlike Bootstrap, we used lambda expression to organize those functions, for ReactJS, we are going to use enum constants.

First, implement the functions to read the year and month.

Organizing functions to read year and month on calendar into integers

We need to implement the functions to read the year and month information displayed on calendar, `ReactCalendarDisplayValue` implements `Function<Browser, Integer>` and we use its constants as the parameter for the constructor of `CalendarPicker` class in Listing [CalendarPicker.java](#).

[ReactCalendarDisplayValue.java](#)

```

import static swb.locators.react.ReactByClassName.CALENDAR;           (5)
import static swb.locators.react.ReactByClassName.DISPLAY_MONTH_YEAR;   (6)

public enum ReactCalendarDisplayValue implements Function<Browser, Integer> {

    DISPLAY_YEAR {
        @Override
        public Integer apply(Browser browser) {
            return parseInt(extract(browser, 1));           (1)
        }
    },
    DISPLAY_MONTH {
        @Override
        public Integer apply(Browser browser) {
            return TO_MONTH.apply(extract(browser, 0)).ordinal();   (2)
        }
    }
};

private static String extract(Browser browser, int i) {   (3)
    return browser.await(CALENDAR)
        .getText(DISPLAY_MONTH_YEAR).split(" ")[i];   (4)
}
}

```

1. Extract year from the display
2. Extract month from the display
3. On the ReactJS datepicker, the month and year are in the same element
4. We need to split it into two, display month and display year
5. The locator enum constant for the calendar
6. The locator enum constant for the display month and year

And ReactJS calendar controls for various buttons.

Organizing ReactJS Calendar Controls

`ReactCalendarControls` implements `Consumer<Browser>` so its constants can be used as the parameter for the constructor of `CalendarPicker` class in Listing [CalendarPicker.java](#). And `CalendarPicker` class is used as the parameters when constructing an instance of `Datepicker` class, as shown in Listing for `BootstrapDatepickerPage`.

[ReactCalendarControls.java](#)

```

import static swb.locators.react.ReactByXpath.TRIGGER_BY; (6)
import static swb.locators.react.ReactByClassName.*; (6)

public enum ReactCalendarControls implements Consumer<Browser> { (6)
    TRIGGER {
        @Override
        public void accept(Browser browser) {
            browser.click(TRIGGER_BY); (1)
        }
    },
    NEXT_MONTH {
        @Override
        public void accept(Browser browser) {
            browser.await(CALENDAR).click(NEXT_MONTH_BUTTON); (2)
        }
    },
    PREVIOUS_MONTH {...}, (3)
    NEXT_YEAR {
        @Override
        public void accept(Browser browser) {
            for (int i = 0; i < 12; i++) { (4)
                NEXT_MONTH.accept(browser);
            }
        }
    },
    PREVIOUS_YEAR {...} (5)
}

```

1. Clicking the trigger to display the calendar
2. Clicking the next month button
3. Clicking the previous month button
4. Since there is no next year button, clicking the next month button 12 times
5. Similar to `NEXT_YEAR`
6. The locator enum constants for ReactJS calendar controls

And day picker for ReactJS.

Implementing ReactJS specific day picking class

ReactJS calendar behaves same as jQuery calendar, it closes after `WebDriver` clicks the day button, so we don't need to do anything to close it.

[ReactDayPicker.java](#)

```
public class ReactDayPicker implements DayPicker {  
  
    private Browser browser;  
  
    @Override  
    public void pick(int day) {  
        browser.await(CALENDAR).findElements(DIV)          (1)  
            .filter(e -> e.getText().equals(String.valueOf(day))) (2)  
            .findFirst()           (4)  
            .get()  
            .click();             (3)  
        browser.await(new ElementVisible(CALENDAR).negate()); (5)  
    }  
}
```

1. Find the calendar and then find all div elements on the calendar
2. Find all `td` elements with the text same as the day string
3. Click it, that is the day we want to pick
4. The first `td` same as the day string is the only one we are looking for
5. Wait for the calendar become invisible

We then add `ReactDatepickerPage` class, just like what we did for jQuery and Bootstrap.

Using page class to apply the Page Object pattern

We add `ReactDatepickerPage` class to encapsulate the `Datepicker` and logic to access elements on the page,

ReactDatepickerPage.java

```
this.datepicker = new Datepicker(  
    new Calendar(browser, TRIGGER),  
    new CalendarPicker(browser, PREVIOUS_YEAR, NEXT_YEAR, DISPLAY_YEAR), (1)  
    new CalendarPicker(browser, PREVIOUS_MONTH, NEXT_MONTH, DISPLAY_MONTH),  
    new ReactDayPicker(browser)  
);
```

1. These constants are from `ReactCalendarControls` and `ReactCalendarDisplayValue`

You can run the following test to see how it works,

ReactDatepickerIT.java

```

@Test
public void pickADate() {
    new ReactDatepickerPage(browser) {
        {
            pick(APRIL, 1, 2015);
            assertEquals("04/01/2015", super.getDate());
        }
    };
}

```

Similarly, we can just copy the structure of those enum from ReactJS package to other packages and modify them to automate other datepickers.

Automating Material-UI datepicker

We can use Web Developer Tool to gather the locators from Material-UI datepicker, unfortunately there is no id can be used so we have to use Xpath locators. We create `MaterialByXpath` enum can add following constants into it.

MaterialByXpath.java

```

TRIGGER_BY("//*[@id='mui-id-2']"),
CALENDAR(
"/html/body/div[2]/div/div[1]/div/div/div[1"],
OK_BUTTON(
"/html/body/div[2]/div/div[1]/div/div/div[2]/button[2]/div/span"),
NEXT_MONTH_BUTTON(
"/html/body/div[2]/div/div[1]/div/div/div[1]/div/div[3]/div[1]/div[3]/button"),
PREV_MONTH_BUTTON(
"/html/body/div[2]/div/div[1]/div/div/div[1]/div/div[3]/div[1]/div[2]/button"),
DISPLAY_MONTH_YEAR(
"/html/body/div[2]/div/div[1]/div/div/div[1]/div/div[3]/div[1]/div[1]/div/div");

```

`MaterialCalendarControls`, `MaterialCalendarDisplayValue`, `MaterialDayPicker` and `MaterialDatepickerPage` look same as the ones for ReactJS implementation except importing the enum constants from `MaterialByXpath`.

```
import static swb.locators.material_ui.MaterialByXpath.*;
```

`MaterialCalendarControls` is the enum with `TRIGGER`, `NEXT_MONTH` and so on.

MaterialCalendarControls.java

```
public enum MaterialCalendarControls implements Consumer<Browser> {      (6)
    TRIGGER {...},
    NEXT_MONTH {...},
    PREVIOUS_MONTH {...},
    NEXT_YEAR {...},
    PREVIOUS_YEAR {...}
}
```

`MaterialCalendarDisplayValue` is the enum with `DISPLAY_YEAR` and `DISPLAY_MONTH`.

MaterialCalendarDisplayValue.java

```
public enum MaterialCalendarDisplayValue implements Function<Browser, Integer> {
    DISPLAY_YEAR {...},
    DISPLAY_MONTH {...};
}
```

`MaterialDayPicker` is not an enum, it is a class and it is almost same as Listing [ReactDayPicker.java](#) expect it need to click "OK" button to close the calendar.

MaterialDayPicker.java

```
public class MaterialDayPicker implements DayPicker {

    @Override
    public void pick(int day) {
        browser.await(CALENDAR).findElements(BUTTON)
            .filter(e -> e.getText().equals(String.valueOf(day)))
            .findFirst()
            .get()
            .click();
        browser.click(OK_BUTTON);                      (1)
        browser.await(new ElementVisible(CALENDAR).negate());
    }
}
```

1. You need to click the "OK" button to close the calendar

And `MaterialDatepickerPage` is almost same as `ReactDatepickerPage` in Listing [ReactDatepickerPage.java](#) except the last parameter is an `MaterialDayPicker`.

MaterialDatepickerPage.java

```

this.datepicker = new Datepicker(
    new Calendar(browser, MaterialCalendarControls.TRIGGER),
    new CalendarPicker(browser, PREVIOUS_YEAR, NEXT_YEAR, DISPLAY_YEAR),
    new CalendarPicker(browser, PREVIOUS_MONTH, NEXT_MONTH, DISPLAY_MONTH),
    new MaterialDayPicker(browser)
);

```

Here is a test to run the datepicker,

MaterialDatepicker_v1_IT.java

```

@Test
public void pickADate() {
    new MaterialDatepickerPage(browser) {
        {
            pick(APRIL, 1, 2015);
            assertEquals("4/1/2015", super.getDate());
        }
    };
}

```

But when you run `MaterialDatepicker_v1_IT` to verify it, it have intermittent error, as shown in following figure,

```

1 test failed - 5s 127ms
=====
21:00:11.412 [main] INFO  swip.tests.TestTimer - =====
21:00:11.414 [main] INFO  swip.tests.TestTimer - Taken 00:00:05.120
21:00:11.414 [main] INFO  swip.tests.TestTimer - =====

org.junit.ComparisonFailure:
Expected :4/1/2015
Actual   :7/1/2015
<Click to see difference>

at org.junit.Assert.assertEquals(Assert.java:115)
at org.junit.Assert.assertEquals(Assert.java:144)
at swip.ch18datepicker.tests.MaterialDatepicker_v1_IT$1.<init>(MaterialDatepicker_v1_IT.java:31)
at swip.ch18datepicker.tests.MaterialDatepicker_v1_IT.pickADate(MaterialDatepicker_v1_IT.java:29) <9 internal calls>

```

Figure 4. Intermittent assertion error

You observe that when Material-UI calendar flips, there is a transition effect so the display month and year flips like a real calendar and some click may misfire so the total flip is not enough even the count is right. And it stops before flipping to target month. So we need to modify `CalendarPicker` to check whether it has reached the month it wants to pick. If not, flip more until the display month is the month we want to pick. It is still like you flip calendar with eyes closed, and when you open eyes to check, you see it is not the month you want to pick, then you flip the calendar until they become same. This logic is still more efficient than checking for every flip.

MaterialByXpath.java

```

void pick(int value) {
    int difference = displayValue.apply(browser) - value;
    while (difference != 0) {           (2)
        if (difference < 0) {
            for (int i = difference; i < 0; i++) {
                next.accept(browser);
            }
        } else if (difference > 0) {
            for (int i = 0; i < difference; i++) {
                previous.accept(browser);
            }
        }
        int newDiff = displayValue.apply(browser) - value;   (1)
        if (difference == newDiff) {
            break;
        }
        difference = newDiff;
    }
}

```

1. This means when it doesn't make any progress when the month or year to pick is out of supported boundary of the calendar
2. Use while loop to flip the calendar until the display month or year is the same as the one to pick

We slightly change it from the previous version, add a `while` loop and another statement by the end of loop the read the calendar and calculate the difference between what's on calendar now and what is the value it needs to pick. If it is not 0, the `while` loop will continue. Normally, it is 0, but due to transition effect, it may not be 0, then the flip the calendar couple of times to reach the month it wants to select.

But when we run the modified test, `MaterialDatePicker_v2_IT`. we get a different error,

```

1 test failed - 4s 813ms
=====
21:14:10.791 [main] INFO  swip.tests.TestTimer - =====
21:14:10.793 [main] INFO  swip.tests.TestTimer - Taken 00:00:04.806
21:14:10.793 [main] INFO  swip.tests.TestTimer - =====
java.lang.ArrayIndexOutOfBoundsException: 1
    at swip.ch18datepicker.material_ui.v1.MaterialCalendarDisplayValue.extract(MaterialCalendarDisplayValue.java:34)
    at swip.ch18datepicker.material_ui.v1.MaterialCalendarDisplayValue.access$100(MaterialCalendarDisplayValue.java:11)
    at swip.ch18datepicker.material_ui.v1.MaterialCalendarDisplayValue$1.apply(MaterialCalendarDisplayValue.java:19)
    at swip.ch18datepicker.material_ui.v1.MaterialCalendarDisplayValue$1.apply(MaterialCalendarDisplayValue.java:16)
    at swip.ch18datepicker.datepicker.v2.CalendarPicker.pick(CalendarPicker.java:30)
    at swip.ch18datepicker.datepicker.v2.Datepicker.pick(Datepicker.java:53)
    at swip.ch18datepicker.material_ui.v2.MaterialDatePickerPage.pick(MaterialDatePickerPage.java:33)
    at swip.ch18datepicker.tests.MaterialDatePicker_v2_IT$1.<init>(MaterialDatePicker_v2_IT.java:30)
    at swip.ch18datepicker.tests.MaterialDatePicker_v2_IT.pickDate(MaterialDatePicker_v2_IT.java:29) <9 internal calls>

```

Figure 5. Intermittent array out of bound exception

We trace the stack trace to this link of code, after split the text, we try to get the second element using index `1` since array starts from `0`.

```
return browser.getText(DISPLAY_MONTH_YEAR).split(" ")[i];
```

Still due to the transition effect, when we split the text, it doesn't have two elements in the result, so it throws the exception,

```
java.lang.ArrayIndexOutOfBoundsException: 1
```

We learnt that in Chapter 6, we can fix the problem by adding wait, so we add one more `await` method with `Function` as parameter to `ExplicitWait`, then the other two methods can just call this new method.

[ExplicitWait.java](#)

```
default Element await(Supplier<By> by) {
    return await((SearchScope e) -> e.findElement(by)); (3)
}

default void await(Predicate<SearchScope> predicate) {
    await((Function<SearchScope, Boolean>) predicate::test); (2)
}

default <T> T await(Function<SearchScope, T> function) {
    return new FluentWait<>(this)
        .withTimeout(1, SECONDS)
        .pollingEvery(10, MILLISECONDS)
        .ignoring(Exception.class)
        .until((SearchScope where) -> function.apply(where)); (1)
}
```

1. `await` method calls the `until` method from `FluentWait`
2. This is "method reference" since Java 8, if you don't cast it to `Function<SearchScope, Boolean>`, there is no compilation error but it will have a StackOverflow exception during runtime
3. Use a lambda expression to construct a `Function` object and call the `await` method which calls `FluentWait`

This method will keep executing the function until it becomes successful or the given time is up. And we can call this method from the `extract` method.

[MaterialCalendarDisplayValue.java](#)

```
private static String extract(Browser browser, int i) {
    return browser.await(
        (SearchScope s) ->
            browser.getText(DISPLAY_MONTH_YEAR).split(" ")[i]); (1)
}
```

1. We wait for the split result having `i + 1` elements

Now `MaterialDatepicker_v3_IT` always passes. You can see, there is extra effort in making Material-UI Date Picker to work, that's the reason we didn't use Material-UI as example in Chapter 17.

Pre-Java 8 alternative

We assume you already become familiar with lambda expression now, but if you don't, here is an example written using anonymous inner class, which is more verbose then the one using lambda expression.

```
private static String extract(final Browser browser,
    final int i) {
    return browser.await(new Function<SearchScope, String>() {
        @Override
        public String apply(SearchScope s) {
            return browser.getText(DISPLAY_MONTH_YEAR).split(" ")[i];
        }
    });
}
```

Next let us look at JsDatePicker datepicker.

Automating JsDatePicker datepicker

We implement `JsDatepickByClassName` for all the locators used by the datepicker.

[JsDatepickByClassName.java](#)

```
CALENDAR("boxMainInner"),
NEXT_MONTH_BUTTON("monthForwardButton"),
PREV_MONTH_BUTTON("monthBackwardButton"),
NEXT_YEAR_BUTTON("yearForwardButton"),
PREV_YEAR_BUTTON("yearBackwardButton"),
DISPLAY_MONTH_YEAR("controlsBarText");
```

As well as `JsDatepickById` for the trigger element.

[JsDatepickById.java](#)

```
TRIGGER_BY("inputField");
```

Unlike other datepickers, JsDatePicker actually has Previous Year and Next Buttons. So instead of clicking Next Month Button 12 times to make a next year equivalent, it can just click the Next Year Button,

[JsDatepickControls.java](#)

```
NEXT_YEAR {
    @Override
    public void accept(Browser browser) {
        browser.await(CALENDAR).click(NEXT_YEAR_BUTTON);           (1)
    }
},
PREVIOUS_YEAR {
    @Override
    public void accept(Browser browser) {
        browser.await(CALENDAR).click(PREV_YEAR_BUTTON);           (2)
    }
}
```

1. It clicks the `NEXT_YEAR_BUTTON` locator

2. It clicks the `PREV_YEAR_BUTTON` locator

The rest of `JsDatepickControls` looks same as the one for ReactJS in Listing [ReactCalendarControls.java](#). The other classes such as `JsDatepickDisplayValue`, `JsDatepickDayPicker` and `JsDatepickPage` look same as the ones for ReactJS except these classes import JsDatePicker specific locators.

```
import static swb.locators.jsdatepick.JsDatepickById.*;
import static swb.locators.jsdatepick.JsDatepickByClassName.*;
```

But if you don't like the repetitive method declaration in this enum, you can change the code to use the following style,

[JsDatepickControlsLambda.java](#)

```

public enum JsDatepickControlsLambda implements Consumer<Browser> {
    TRIGGER(browser -> browser.click(TRIGGER_BY)),      (2)
    NEXT_MONTH(browser -> browser.await(CALENDAR).click(NEXT_MONTH_BUTTON)),
    PREVIOUS_MONTH(browser -> browser.await(CALENDAR).click(PREV_MONTH_BUTTON)),
    NEXT_YEAR(browser -> browser.await(CALENDAR).click(NEXT_YEAR_BUTTON)),
    PREVIOUS_YEAR(browser -> browser.await(CALENDAR).click(PREV_YEAR_BUTTON));

    private Consumer<Browser> consumer;   (1)

    JsDatepickControlsLambda(Consumer<Browser> consumer) {   (3)
        this.consumer = consumer;
    }

    @Override
    public void accept(Browser browser) {
        consumer.accept(browser);   (4)
    }
}

```

1. This enum has a `Consumer<Browser>` instance variable.
2. The lambda expression creates an instance of `Consumer<Browser>` and call the constructor of this enum `JsDatepickControlsLambda`
3. The constructor of this enum `JsDatepickControlsLambda`
4. Delegate the call to the `Consumer<Browser>` instance variable

You can see it removed the duplicated method declaration `@Override public void accept(Browser browser)` from each constants in Listing [JsDatepickControls.java](#) and only have one enum level `accept` method. But it may not be worth to do the same thing for `JsDatepickDisplayValue`. The more constants in the enum, the more worthwhile to do it, otherwise, it is just a personal preference to use either way.

Since it has year buttons, we actually can write some cool tests. [2]

[BackToFutureUsingJsDatepickIT.java](#)

```

@Test
public void backToFuture1() {
    new JsDatepickPage(browser) {
        {
            pick(NOVEMBER, 5, 1955);                                (1)
            assertEquals("05-NOV-1955", super.getDate());
        }
    };
}

@Test
public void backToFuture2() {
    new JsDatepickPage(browser) {
        {
            pick(OCTOBER, 21, 2015);                                (2)
            assertEquals("21-OCT-2015", super.getDate());
        }
    };
}

@Test
public void backToFuture3() {
    new JsDatepickPageLambda(browser) {
        { (4)
            pick(SEPTEMBER, 2, 1885);                            (3)
            assertEquals("02-SEP-1885", super.getDate());
        }
    };
}

```

1. This is the date Marty McFly met his parents, George and Lorraine while they were teenage.
2. This is the date Marty McFly met his future son, Marty Jr.
3. This is the date Marty McFly met his great-great-grandparents, Seamus and Maggie McFly
4. This test use the enum written with lambda expression

Is it awesome? JsDatePicker is one of earliest datepickers available for people to use and it is the only one in the example with the year buttons. You can see it takes you back to 1885 in 9 seconds. Material-UI Date Picker is the fanciest datepicker from UI perspective. But if they don't add year buttons, anyone plans to build a time machine, they should consider to choose JsDatePicker.

You have seen we use the same framework in the automation of 5 kinds of datepickers and they all work.

Summary

- Extracting general purpose datepicker framework from the single class
 - Provide a jQuery specific datepicking functions for the framework
 - Automating Bootstrap datepicker using the framework with Bootstrap specific datepicking implementation
 - Automating ReactJS, Material-UI and JsDatePick datepickers
-

1. <https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html>

2. These dates used in the tests are the destination dates of the time travels set on Lorraine, the Time Machine, from movie trilogy Back to Future.

Appendices

Appendix A: Selenium Grid

This appendix covers

- An overview of the architecture of the grid
- Local, vs **on premise** vs **Selenium as a service**
- Using the `RemoteWebDriver` class
- Using Vagrant to learn about Selenium Grid

Selenium Grid is a way to run test using browsers you do not have installed on your local machine, or on other operating systems. Instead of starting the browser on your local machine, the browser is started on a remote machine that is part of the grid. Your tests run on your local machine, but connect to a process on the remote machine. That process opens the browser, which in turn opens pages on the web application you are testing.

The key components for running a test script on a grid are:

- A test script – of course! This might run on your local machine, or a CI server.
- A number of **Nodes**, each running on different operating systems and with different browsers.
- The **Hub** application running on a single **hub machine**. This keeps a track of the nodes, and proxies requests to them.
- The web application you are testing.

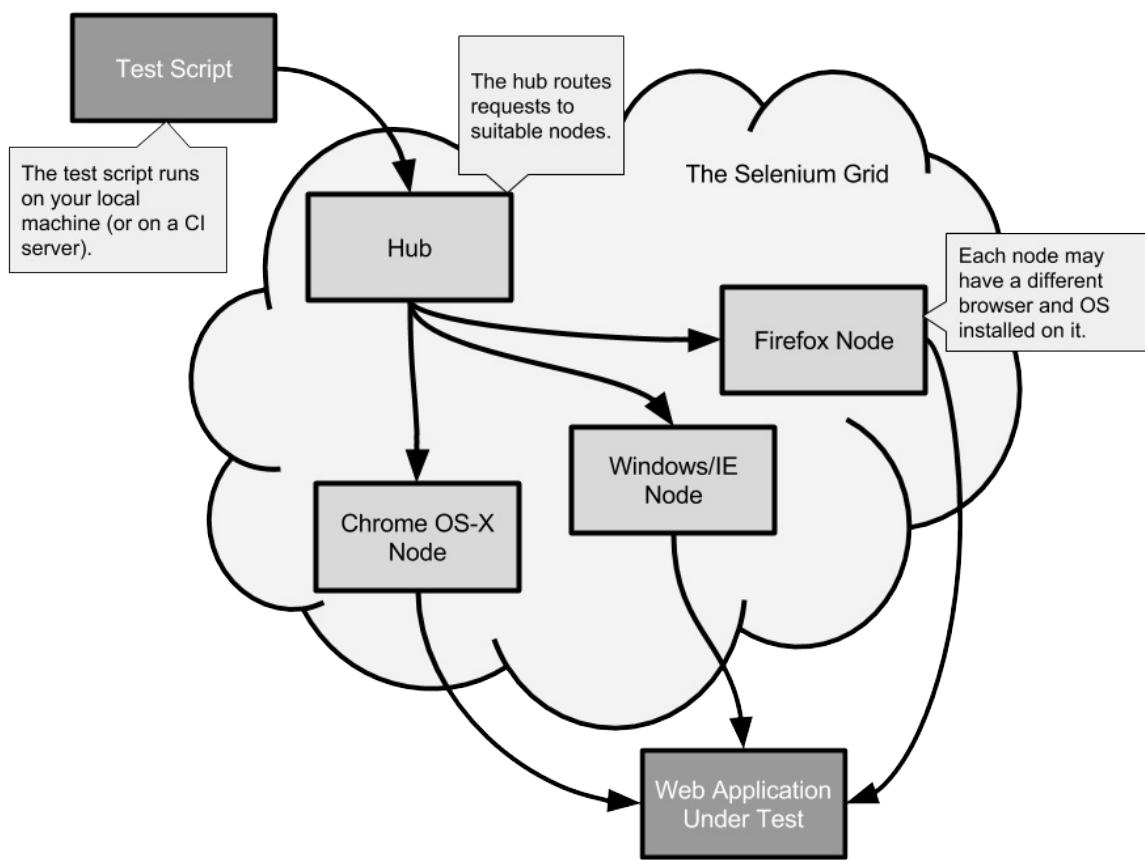


Figure 1. Selenium Grid

There's a number of pros and cons to using a grid.

Pros:

- It will give you a much wider range of browsers and operating systems to test on.
- Problems with your local machine, or the browser install won't affect your test.
- You can run tests in parallel, so your test suite will finish quicker and you can get faster feedback.
- If the browser or the remote machine crash, it will not affect your machine, or other tests.

Cons:

- As the browser is on another machine, you cannot see the test running to help you understand the cause of any problems with the tests.
- The machines within the grid must be able to access your web application.
- You need someone skilled to run and maintain the grid.

You've two options if you want to use a grid. You can use a **Selenium as a service** solution, such as Saucelabs (<https://saucelabs.com>), or BrowserStack (<https://www.browserstack.com>), or you can build an **on premise solution**. There are different costs associated with each, and we'll talk about this shortly.

By the end of this appendix you will have learnt about the architecture of a grid; some of the differences between local, on premise, and Selenium as a service solutions; and how to run a local grid using Vagrant that will help you learn about Grid.

Locally vs "Selenium as a service" vs on premise solutions

There are various pros and cons of using each way of running testing using WebDriver.

You can have your browser running locally on your local machine. This is good for:

- Getting started quickly
- Writing new tests
- Debugging failing tests
- Experimenting, or trying new features out
- When you are only working on a single platform, browser, or device
- If you only have small suite, that can be run in sequence

You can have your browsers running on an on premise grid. To do this, you would set up and run the grid on your company's premises. This is good for:

- Learning about Selenium Grid
- When you have a large test suite
- If you have some special browser or devices that third-party solutions do not support
- When you do not wish to expose your application to third-parties, e.g. due to security requirements
- When you and your team are experienced at running networks of machines similar to a grid

You can have your browser running on a grid maintained by an external company – Selenium as a service. This is good for:

- When you have a large test suite
- If you need to test on browsers and devices your company doesn't own
- Saving time and effort maintaining software and hardware needed for a grid
- When you don't want to have to purchase licenses
- Having features set-up for you
- Clearly set out costs

This is not to say you can't use a combination of each of these solutions. You'll probably find yourself developing your tests on your local machine, and using a grid for running the test suites on your CI system.

Architecture

A **Selenium Grid** contains a number of machines. Most of the machines undertake the role of **nodes** (their job is to run the browser), and one machine who undertakes the role of the **hub** (who's job it is to keep track of the nodes). You'll also need a machine to run your tests on (this might be your CI server), and one or more machines where the **web applications** you are testing are running (Web Application Under Test or **WAUT**).

Hub Failure

Your hub maybe a single point of failure within your grid. Make sure it's easy to replace, and you have a process in place to do this.

When a node starts up, it connects to the hub to tell the hub that it is available to run tests. It also tells the hub what browsers are installed and what operating system it is. Finally, it tells the hub to send traffic to a certain port (5555 by default). The hub keeps a list of nodes attached to it. When a test script connects to the hub, the script can request a browser based on the desired capabilities. The hub then chooses a node based on the request. For example, if you need an Internet Explorer node, then only nodes with IE installed would be used.

Each node may only run a limited number of browsers at once. For example, they may only be able to run 5 Firefox browsers. This means up to five tests can run in parallel. The hub locks the browser to your test, so no one else can use it.

Then, when your test script sends a request to the hub to open a page, click on an element, etc, the request is forwarded to the appropriate node which then executes them and returns the results. The hub acts as a proxy for your requests.

Finally, when your test closes the browser, the node will close the browser, and the lock is released so that other tests can use it.

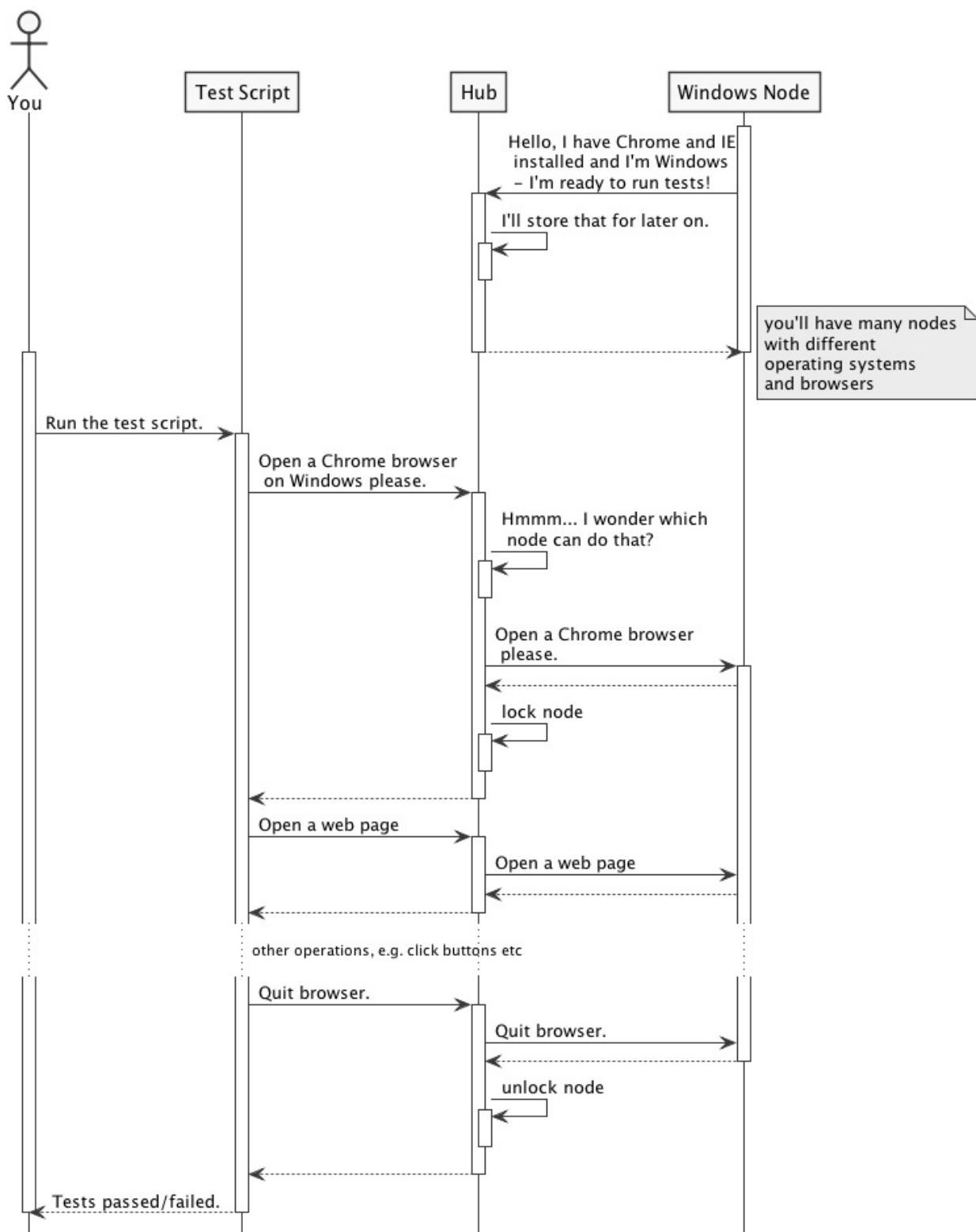


Figure 2. Selenium Grid Sequence

Using the `RemoteWebDriver` class

To use a grid, rather than use `FirefoxDriver` (or whichever driver you normally use), you must use `RemoteWebDriver`. This driver takes two arguments. The first argument is a URL to the grid's hub, the second is capabilities you want your browser to have, such as which browser it is (e.g. Chrome) or which operating system (e.g. Windows or OS-X).

```
new RemoteWebDriver(remoteUrl, desiredCapabilities)
```

The `remoteUrl` is usually in the form of `http://yourusername:yourpassword@yourhub/wd/hub`. For the capabilities, you can specify the browser, version and OS.

```
DesiredCapabilities desiredCapabilities = DesiredCapabilities.firefox();
desiredCapabilities.setCapability("version", "7");
desiredCapabilities.setCapability("platform", Platform.XP);
```

When running a `RemoteDriver`, you almost certainly want to want to wrap the driver in an `Augmenter` object. An `Augmenter` adds the ability to take screenshots to the driver, very useful if you cannot see the browser!

```
new Augmenter().augment(webDriver)
```

If your test use a specific concrete browser class (e.g. `ChromeDriver`) in your test, this will need to change your tests to use the `WebDriver` interface.

Running the code in the book on a grid

The code that comes with this book supports remote driver out of the box. You can run against a remote driver by setting these system properties:

- `webdriver.remote=true`
- `webdriver.remote.url=http://hub:4444/wd/hub`

And optionally, capabilities:

- `webdriver.capabilities.browserName=firefox`
- `webdriver.capabilities.platform=windows`
- `webdriver.capabilities.version=7`

For example:

```
mvn install -Dwebdriver.remote=true  
-Dwebdriver.remote.url=http://hub:444/wd/hub  
-Dwebdriver.capabilities.browserName=chrome
```

You can run all tests in the book any any browser or operating system, locally or remotely.

If you are running a test application locally, you cannot use `http://localhost:8080` or `http://127.0.0.1:8080` as the base URL. You should use the host name (or IP address) of your machine.

You can find out the hostname on Linux and OS-X by running the `hostname` command.

Running a Selenium Grid

We've provided a sample of running a small grid using Vagrant on your local machine with this book's code. This will give you a chance to experiment with a grid and learn the concepts.

Vagrant is a tool for managing virtual machines. It's useful with grid, as you can run several virtual machines on your local machine. This is perfect for learning how grids work.

If you've not used Vagrant before, we strongly recommend you take some time to learn more about it on their web site (<https://www.vagrantup.com>) before reading the rest of this section. Naturally, you'll need to install it as well.

We've provided a `Vagrantfile` with the book's source code. This file can be used to start-up a local grid that includes:

- A hub.
- An Ubuntu node with Firefox.
- Another Ubuntu node with Chrome.
- A Windows 8 node running Internet Explorer.

```
cd vagrant  
vagrant up
```

The Ubuntu nodes have a set-up script, you will (of course) have to set-up the Windows node manually as detailed below.

To start with you need a computer to run the hub machine.

On your hub machine, to start the hub application you need to do the following:

1. Install Java.
2. Download the standalone server JAR (e.g. `selenium-server-standalone-2.48.2.jar`) from <http://www.seleniumhq.org/download/>.
3. In a command prompt, run:

Starting Selenium Hub

```
java -jar selenium-server-standalone-2.48.2.jar -role hub
```

Naturally, you should use the latest version number. You should then check the hub is working. You should see something similar to the following printed on the console:

Logs Of A Successful Hub Start-up

```
13:04:39.077 INFO - Launching Selenium Grid hub
...
13:04:40.087 INFO - Nodes should register to http://192.168.10.2:4444/grid/register/
13:04:40.087 INFO - Selenium Grid hub is up and running
```

The URL logged is useful, it is the URL you must configure your nodes to connect to. The IP might change if you reboot your hub. You should check is it visible at <http://192.168.10.2:4444/>. You should see the homepage as per figure [Homepage](#):

You are using grid 2.48.0
Find help on the official selenium wiki : [more help](#)
default monitoring page : [console](#)

Figure 3. Homepage

To start a node you need to follow steps 1 and 2 above. Install any browsers you need, and as your command run:

Starting A Selenium Node

```
java -jar selenium-server-standalone-2.48.2.jar -role node -hub http://192.168.10.2:4444/g
rid/register
```

You'll need to set the IP to your hub's IP. You should check this is working, the console should show the following:

Logs Of A Successfully Started Node

```
13:18:46.841 INFO - Launching a Selenium Grid node
13:18:47.608 INFO - Java: Oracle Corporation 24.91-b01
13:18:47.608 INFO - OS: Linux 3.2.0-23-generic amd64
...
13:18:47.734 INFO - Selenium Grid node is up and ready to register to the hub
13:18:47.773 INFO - Starting auto registration thread. Will try to register every 5000 ms.
13:18:47.774 INFO - Registering the node to the hub: http://192.168.10.2:4444/grid/register
13:18:47.814 INFO - The node is registered to the hub and ready to use
```

You can also look in the hub logs to see if a node has registered with it:

Node Being Registered In Hub Logs

```
13:18:47.842 INFO - Registered a node http://192.168.10.3:5555
```

The IP listed is that of the node.

If you return to the hub web application, and open the console <http://192.168.10.2:4444/grid/console>, you should see the following:



Figure 4. Hub Console

If you see `Connect to 192.168.10.5:5555 [/10.0.2.15] failed: Connection refused` then you might have to modify the machine's firewall to allow the hub to connect to the node. You can test if it is working by opening <http://192.168.10.5:5555/wd/hub>, you should

seen information about the node.

Finally, you should then configure your tests to use the hub's URL, for example

<http://192.168.10.2:444/wd/hub>

Now, your grid is set-up to run.

Summary

In this appendix you have learnt about Selenium Grid. A grid will allow you to test faster, on a greater variety of browsers and operating systems. We looked at running a grid locally, on premises or using a third-party's "Selenium as a service". Each of these has some benefits and some trade offs, and you may find yourself using a combination of them. Setting up a grid can be a time consuming task, so you may want to have a discussion with your team to make sure that the benefits are greater than the costs.