Web technologies are becoming increasingly complex and there is often a need to test your web applications against a vast number of browsers and platforms, so you need to build highly reliable and maintainable test automation cases. Selenium is one of the most easy to implement and efficient solutions for this.

This book will guide you through designing and implementing the automation framework on Selenium to build advanced automated test cases. You will start with getting acquainted with the Selenium IDE, working with AJAX, and using different methods to locate elements in a web page. You will then move on to using PageObjects, making tests maintainable, using WebDriver with different browsers, and setting up Selenium Grid. You will also be introduced to advanced topics such as working with proxy servers, network intrusions, and more. By the end of the book, you will have the skills to efficiently test your web applications using Selenium. A well-detailed section has been dedicated to teaching you the object-oriented concepts with basics of core Java, which are the pre-requisites for Selenium automation.

Who this book is written for
If you are a software developer with a basic knowledge of testing and are interested in automated testing using Selenium, this is the book for you. No prior knowledge of Selenium is required.

What you will learn from this book
- Understand designing and implementing the automation framework
- Understand and implement AJAX in your web pages
- Set up Selenium WebDriver in both IntelliJ and Eclipse
- Build test suites in Selenium using PageObjects
- Get to know about WebElement handling with Selenium WebDriver
- Install Selenium WebDriver for mobile devices
- Understand and learn testing in Selenium Grid

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Selenium IDE'
- A synopsis of the book’s content

About the Author

Raghavendra Prasad MG is basically an electronics and communication engineer from the University of Mysore and did his MBA from VMU. He is currently working as an automation architect and a QA lead at Riversand Global Technologies. He has been into test automation for more than 8 years. He is an automation tools and automation framework design trainer as well. He has trained at least 500 professionals so far. In his past experience, he worked as a corporate trainer on automation tools and automation framework design.

His previous experiences include Life Technologies (now Thermo Fischer Scientific Company), GT Nexus Software, Mphasis (an HP Company), Thomson-Reuters Inc., Telelogic (now IBM-Rational), Aztecsoft (now Mindtree Consulting Ltd), and many more.

He worked as a freelancer for automation and automation framework implementation.

He implemented various automation frameworks for more than 30 projects, starting from zero level on applications of multiple domains such as, ERP, CRM, ELM, financial domain, automobile, insurance and pension (on legacy systems), healthcare, biotechnology, cloud supply chain platform, and Master Data Management (MDM).

Raghu is especially well known for enabling the manual (or functional) tester towards automation. His experience in this area has become a key point for this book. Raghu has built his own tools on API automation, BI automation, and performance.
Raghu won an award for being the best performer in the keyword test automation framework design and implementation at GTNexus. He also won an Individual Summit award for Data Driven Test Automation Framework design and implementation at MphasiS, an HP Company for GM (General Motors, a supply chain project).

Raghu won an Excellence in Execution award at Telelogic, an IBM Company. He is passionate about test automation. You can reach him at his LinkedIn account, https://in.linkedin.com/pub/raghavendra-prasad-mg/4/538/264.
Learning Selenium Testing Tools
Third Edition

Selenium WebDriver is the most used automation tool for web-based applications. This book shows beginners, developers, and testers how to create automated test cases using Selenium. You will be able to use Selenium IDE for quick throwaway tests and you will learn to use Selenium WebDriver along with the automation framework development.

You will learn how to use Selenium WebDriver with both desktop browsers and mobile browsers as well learn good design patterns to ensure your tests will be extremely maintainable.

We have provided prerequisites as well, so that you are well versed with the topics needed to understand testing with Selenium. You will therefore not have to refer to other external sources to understand these topics.

What This Book Covers

Chapter 1, Getting Started with Selenium IDE, explains how to install Selenium IDE and record our first tests. We will see what is needed to work against AJAX applications.

Chapter 2, Locators, shows how we can find elements on the page to be used in our tests. We will use XPath, CSS, link text, and ID to find elements on the page so that we can interact with them.

Chapter 3, Overview of the Selenium WebDriver, discusses all the history and architectural designs for Selenium WebDriver. You will also go through the necessary items for setting up a development environment.

Chapter 4, Finding Elements, explains all the different techniques in searching elements using Selenium WebDriver. This chapter builds on the locators that we learned in Chapter 2, Locators.

Chapter 5, Design Patterns, introduces the different design patterns that can be used with Selenium WebDriver. The design patterns will show you how to make your tests more maintainable and allow more people to work on your code.

Chapter 6, Working with WebDriver, introduces all the different aspects of getting different browsers that Selenium WebDriver supports on desktop operating systems.

Chapter 7, Automation Framework Development and Building Utilities, explains the various types of automation frameworks and how to build the Hybrid automation framework with the design, architecture, and implementation. We are giving a plug and play Hybrid automation framework for Selenium WebDriver with the code bundle of this book.
Chapter 8, *Mobile Devices*, explains how Selenium WebDriver works on mobile devices to test mobile websites or sites built with responsive web design.

Chapter 9, *Getting Started with the Selenium Grid*, shows us how we can set up our Selenium Grid. We will also take a look at running tests in parallel to try bringing down the time it takes to run tests.

Chapter 10, *Advanced User Interactions*, explains how to build chains of actions together to help when you need to drag and drop or have key combinations working. We will also look at how we can press a mouse button and hold it down while we move the mouse.

Chapter 11, *Working with HTML5*, explains working with some of the HTML5 technologies that are becoming available to browsers. The Selenium WebDriver APIs are very similar to the JavaScript APIs in the browser in order to make use of them easier.

Chapter 12, *Advanced Topics*, explains how to capture network traffic between the browser and the web server. We finish off by capturing screenshots.

Chapter 13, *Migrating from Remote Control to WebDriver*, introduces how the interaction with the browser has changed and how we can convert our Selenium 1 tests to Selenium 2 to take advantage of the changes in Selenium WebDriver.

Appendix A, *Automation Prerequisites for Selenium Automation*, deals with the basics of Java and prerequisites related to programming, which is mandatorily required for automation testing. We are giving an automation prerequisites-related program of examples with this book.

Appendix B, *Answers for Self-test Questions*, has the answers to all the self-test questions in the previous chapters.
Test automation has grown in popularity over the years because developers do not have the time or money to invest in large test teams to make sure that applications work as they are expected. Developers also want to make sure that the code they have created works as they expect it to.

Jason Huggins saw this issue too and wanted to make sure that any system he was working on would work on multiple operating systems and browsers. So he created Selenium.

Selenium IDE is a fully-featured IDE (Integrated Development Environment) that installs as a plugin in Mozilla Firefox and enables developers to test their web applications through Selenium. With the Selenium IDE, you can record user interactions with the web browser and play them back to test for errors. It's a powerful, robust IDE that radically simplifies and automates the QA testing process.

Selenium is one of the most well-known testing frameworks in the world that is in use. It is an open source project that allows testers and developers alike to develop functional tests to drive the browser. It can be used to record workflows so that it reduces the time in regression testing. Selenium can work on any browser that supports JavaScript, since Selenium has been built using JavaScript.

In this chapter, you will learn the basics of the Selenium IDE, how to use it, and how to locate a web element on a web page. We shall cover the following topics:

- What is Selenium IDE
- Recording our first test
- Updating tests to work with AJAX sites
Getting Started with Selenium IDE

- Using variables in our tests
- Debugging tests
- Saving tests to be used later
- Creating and saving test suites

So, let's get on with it...

Before we start working through this chapter, we need to make sure that Mozilla Firefox is installed on your machine. If you do not have Mozilla Firefox installed, you will need to download it from http://www.getfirefox.com/.

Understanding Selenium IDE

Selenium IDE is a Firefox add-on developed originally by Shinya Kasatani as a way to use the original Selenium Core code without having to copy Selenium Core onto the server. Selenium Core is the key JavaScript module that allows Selenium to drive the browser. It has been developed using JavaScript so that it can interact with DOM (Document Object Model) using native JavaScript calls.

Selenium IDE was developed to allow testers and developers to record their actions as they follow the workflow that they need to test.

Installing Selenium IDE

Now that we understand what Selenium IDE is, it is a good time to install it. At the end of these steps, you will have successfully installed Selenium IDE onto your computer:

2. Click on the download link for Selenium IDE. You may see a message appear saying Firefox prevented this site (seleniumhq.org) from asking you to install software on your computer. If you do, click the Allow button.
3. A Firefox prompt will appear, as shown in the following screenshot:

![Firefox prompt](image)

4. You will then be asked if you would like to install Selenium IDE and the exporter add-ons. These have been made pluggable to the IDE by the work that Adam Goucher did. You will see a screenshot similar to the following one:

![Software Installation](image)

5. Click on **Install** button. This will now install Selenium IDE and formatters as Firefox add-ons.
Getting Started with Selenium IDE

6. Once the installation process is complete, it will ask you to restart Firefox. Click the **Restart Now** button. Firefox will close and then reopen. If you have anything open in another browser, it might be worth saving your work as Firefox will try to go back to its original state. However, this cannot be guaranteed.

7. Once the installation is complete, the add-ons window will show the Selenium IDE and its current version:

You have successfully installed Selenium IDE and we can start thinking about writing our first test.

**Getting acquainted with the Selenium IDE tool**

Now that Selenium IDE has been installed, let's take some time to familiarize ourselves with it. This will give us the foundation that we can use in later chapters. Open up Selenium IDE by going through the tools menu in Mozilla Firefox. Navigate to **Tools | Selenium IDE**. A window will appear. If the menu bar is not available, which is now the default in Firefox, you can launch Selenium IDE via **Firefox | Web Developer | Selenium IDE**.
Starting from the top, I will explain what each of the items are:

- **Base URL**: This is the URL that the test will start at. All open commands will be relative to Base URL unless a full path is inserted in the open command.
- **Speed Slider**: This is the slider under the Fast and Slow labels in the screenshot.
- **Play** icon: This play entire test suite icon runs all the tests in the IDE.
- **Play** icon: This play current test case icon runs a single test in the IDE.
- **Pause/Resume** icon: This pause/resume icon pauses a test that is currently running.
- **Step** icon: This step icon steps through the test once it has paused.
- **Record** icon: This is the record button. This will be engaged when the test is recording.
• The **Command** drop-down list has a list of all the commands that are needed to create a test. You can type into it to use the autocomplete functionality or use it as a dropdown.

• The **Target** textbox allows you to input the location of the element that you want to work against.

• The **Find** button, once the target box is populated, can be clicked to highlight the element on the page.

• The **Value** textbox is where you place the value that needs to change. For example, if you want your test to type in an input box on the web page, you will put what you want it to type in the value box.

• The **Table** tab will keep track of all your commands, targets, and values. It has been structured this way because the original version of Selenium was styled on FIT tests. **FIT (Framework for Integrated Testing)** was created by Ward Cunningham. The tests were originally designed to be run from HTML files and the IDE keeps this idea for its tests.

• If you click the **Source** tab, you will be able to see the HTML that will store the test. Each of the rows will look like this:

```html
<tr>
  <td>open</td>
  <td>/chapter1</td>
  <td></td>
</tr>
```

• The area below the **Value** textbox will show the Selenium log while the tests are running. If an item fails, then it will have an `[error]` entry. This area will also show help on Selenium commands when you are working in the **Command** drop-down list. This can be extremely useful when typing commands into Selenium IDE instead of using the record feature.

• The **Log** tab will show a log of what is happening during the test. The **Reference** tab gives you documentation on the command that you have highlighted, and is also useful if you forgot some command; users can just start writing command in the **Command** field, then select **some like searched** and read the reference.
Rules in creating tests with Selenium IDE

Now that we have installed Selenium IDE and understand what it is, we can think about working through our first tests. There are a few things that you need to consider when creating your first test. These rules apply to any form of test automation but need to be adhered to, especially when creating tests against a user interface:

- Tests should always have a known starting point. In the context of Selenium, this can mean opening a certain page to start a workflow.
- Tests should not have to rely on any other tests to run. If a test is going to add something, do not have a separate test to delete it. This is to ensure that if something goes wrong in one test, it will not mean you have a lot of unnecessary failures to check.
- Tests should only test one thing at a time.
- Tests should clean up after themselves.

These rules, like most rules, can be broken. However, breaking them can mean that you may run into issues later on, and when you have hundreds, or even thousands of tests, these small issues can mean that large parts of a test suite are failing.

With these rules in mind, let's create our first Selenium IDE test.

Recording your first test with Selenium IDE

We will record our first test using Selenium IDE. To start recording the tests, we will need to start Mozilla Firefox. Once it has been loaded, you will need to start Selenium IDE. You will find it under the Tools drop-down menu in Mozilla Firefox or in the Web Developer drop-down menu. Once loaded, it will look like the next screenshot. Note that the record button is engaged when you first load the IDE.

To start recording your tests, let's do the following:

2. On the web application, do the following:
   1. Click on the record button (red-colored radio button).
   2. Select another value from the drop-down box, for example, Selenium RC.
3. Click on the **Home Page** link.

![Selenium IDE](image)

3. Your test has now been recorded and should look like the preceding screenshot.

4. Click the play button that looks like this icon: 

![Play button](image)
5. Once your test has completed, it will look like the following screenshot:

![Screenshot of Selenium IDE test cases]

We have successfully recorded our first test and played it back. As we can see, Selenium IDE tried to apply the first rule of test automation by specifying the `open` command. It set the starting point of the test, in this case, `/chapter1`, and then it began stepping through the workflow that we want to record.

Once the actions have all been completed, you will see that all of the actions have a green background. This shows that they have completed successfully. On the left, you will see that it has completed one successful test, or run, within Selenium IDE. If you were to write a test that failed, the `Failure` label will have a 1 next to it.

**Validating a test with `assert` and `verify`**

In the last few steps, we were able to record a workflow that we would expect the user to perform. It will test that the relevant bit of functionality is there, such as buttons and links to work against. Unfortunately, we are not checking whether the other items on the page are there or if they are visible when they should be hidden. We will now work against the same page as before, but we shall make sure that different items are on the page.
Getting Started with Selenium IDE

There are two mechanisms for validating elements available on the application under test. The first is **assert**: this allows the test to check whether the element is on the page. If it is not available, then the test will stop on the step that failed. The second is **verify**: this allows the test to check if the element is on the page, but if it isn't, then the test will carry on execution. To add the assert or verify commands to the tests, we need to use the context menu that Selenium IDE adds to Firefox. All that one needs to do is right-click on the element if on Windows or Linux. If you have a Mac, then you will need to do the two-finger click to show the context menu.

When the context menu appears, it will look roughly like the following screenshot with the normal Firefox functions above it:

![Context Menu Screenshot]

We will record the test and learn how to use/verify some commands as follows:

1. Open the IDE so that we can start recording.
3. Select **Selenium Grid** from the drop-down box.
4. Change the selection to **Selenium Grid**.
5. Verify that the **Assert that this text is on the page** text is mentioned on the right-hand side of the drop-down box, by right-clicking on the text and selecting **verifyText id=diveontheleft Assert that this text is on the page**. You can see the command in the previous screenshot.
6. Verify that the button is on the page. You will need to add a new command for **verifyElementPresent** with the **verifybutton** target in Selenium IDE.
7. Now that you have completed the previous steps, your Selenium IDE should look like the following screenshot:

If you now run the test, you will see it has verified that what you are expecting to see on the page has appeared. Notice that the verify commands have a darker green color. This is to show that they are more important to the test than moving through the steps. The test has now checked that the text we required is on the page and that the button is there too.
Getting Started with Selenium IDE

What will happen if the verify command did not find what it was expecting? The IDE would have thrown an error stating what was expected was not there, but then carried on with the rest of the test. We can see an example of this in the following screenshot:

![Selenium IDE screenshot](image)

The test would not have carried on if it was using assert as the mechanism for validating that the elements and text were loaded with the page.

We have just seen that we can add asserts or verification to the page. Selenium IDE does not do this when recording, so it will always be a manual step. The assert command will stop the running tests, unlike the verify command in which the tests will continue running even after failure. In both the cases, Selenium IDE will log an error. Each of these have their own merits.

Some of the verify and assert methods are as follows:

<table>
<thead>
<tr>
<th>Selenium Command</th>
<th>Command Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>verifyElementPresent</td>
<td>This verifies an expected element is present on the page.</td>
</tr>
<tr>
<td>assertElementPresent</td>
<td>This asserts an expected element is present on the page.</td>
</tr>
<tr>
<td>verifyElementNotPresent</td>
<td>This verifies an expected element is not present on the page.</td>
</tr>
<tr>
<td>Selenium Command</td>
<td>Command Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>assertElementNotPresent</td>
<td>This asserts an expected element is not present on the page.</td>
</tr>
<tr>
<td>verifyText</td>
<td>This verifies expected text and its corresponding HTML tags are present on the page.</td>
</tr>
<tr>
<td>assertText</td>
<td>This asserts expected text and its corresponding HTML tags are present on the page.</td>
</tr>
<tr>
<td>verifyAttribute</td>
<td>This verifies the value of an attribute of any HTML tag on the page.</td>
</tr>
<tr>
<td>assertAttribute</td>
<td>This asserts the value of an attribute of any HTML tag on the page.</td>
</tr>
<tr>
<td>verifyChecked</td>
<td>This verifies whether the condition of the checkbox is checked or not on the page.</td>
</tr>
<tr>
<td>assertChecked</td>
<td>This asserts whether the condition of the checkbox is checked or not on the page.</td>
</tr>
<tr>
<td>verifyAlert</td>
<td>This verifies the alert present on the page.</td>
</tr>
<tr>
<td>assertAlert</td>
<td>This asserts the alert present on the page.</td>
</tr>
<tr>
<td>verifyTitle</td>
<td>This verifies an expected page title.</td>
</tr>
<tr>
<td>assertTitle</td>
<td>This asserts an expected page title.</td>
</tr>
</tbody>
</table>

**Creating comments in your tests**

Before we carry on further with Selenium, this is a good time to mention how to create comments in your tests. As all good software developers know, having readable code and having comments can make maintenance in the future much easier. Unlike in software development, it is extremely hard, almost impossible, to write self-documenting code. To combat this, it is good practice to make sure that your tests have comments that future software testers can use.

**Adding Selenium IDE comments**

To add comments to your tests, perform the following steps:

1. In the test that was created earlier, right-click on a step. For example, the verify step.
2. The Selenium IDE context menu will be visible as shown in the following screenshot:

3. Click on **Insert New Comment**. A space will appear between the Selenium commands.

4. Click on the **Command** textbox and enter in a comment so that you can use it for future maintenance. It will look like the following screenshot:
We just had a look at how to create comments. Comments will always appear as purple text in the IDE. This, like in most IDEs, is to help you spot comments quicker when looking through your test cases. Now that we know how to keep our tests maintainable with comments, let’s carry on working with Selenium IDE to record/tweak/replay our scripts.

**Multiplying windows**

Web applications, unfortunately, do not live in one window of your browser. An example of this can be a site that shows reports. Most reports will have their own window so that people can easily move between them.

Unfortunately, in testing terms, this can be quite difficult to do, but in this section, we will have a look at creating a test that can move between windows.

**Working with multiple windows**

Working with multiple browser windows can be one of the most difficult things to do within a Selenium test. This is down to the fact that the browser needs to allow Selenium to programmatically know how many child browser processes have been spawned.

In the following examples, we will see the tests click on an element on the page that will cause a new window to appear. If you have a pop-up blocker running, it's a good idea to disable it for this site while you work through these examples.

Open up Selenium IDE and go to the Chapter 1 page on the site and refer to the following steps:

1. Click on one of the elements on the page that has the text Click this link to launch another window. This will cause a small window to appear.
2. Verify the text in the popup by right-clicking and selecting VerifyText id=popup text within the popup window.
3. Once the window has loaded, click on the Close the Window text inside it.
4. Add a verify command for an element on the page. Your test should now look like the following screenshot:

Sometimes, Selenium IDE will add a `clickAndWait` command instead of a `click` command. This is because it notices that the page has to unload. If this happens, just change the `clickAndWait` command to a `click` so that it does not cause a timeout in the test.

In the test script, we can see that it has clicked on the item to load the new window and then has inserted a `waitForPopUp` command. This is so that your test knows that it has to wait for a web server to handle the request and the browser to render the page. Any commands that require a page to load from a web server will have a `waitFor` command. The next command is the `selectWindow` command. This command tells Selenium IDE that it will need to switch context to the window, called `popupwindow`, and will execute all the commands that follow in that window unless told otherwise by a later command.
Once the test has finished with the pop-up window, it will need to return to the parent window from where it started. To do this, we need to specify null as the window. This will force the `selectWindow` command to move the context of the test back to its parent window.

**Complex working with multiple windows**

In this example, we will open two pop-up windows and move between them and the parent window as it completes its steps:

1. Start Selenium IDE and go to **Chapter 1** on the website.
2. Click on the **Click this link to launch another window** link. This will launch a pop-up window.
3. Assert the text on the page. We do this by right-clicking and selecting **assertText**.
4. Go back to the parent window and click on the link to launch the second pop-up window.
5. Verify the text on the page.
6. Move to the first pop-up window and close it using the close link. As before, be aware of `clickAndWait` instead of `click`.
7. Move to the second pop-up window and close it using the close link.
8. Move back to the parent window and verify an element on that page.
9. Run your test and watch how it moves between the windows. When complete, it should look like the following screenshot:

![Selenium IDE Test Runner](image)

We just had a look at creating a test that can move between multiple windows. We saw how we can move between the child windows and its parent window as though we were a user.
Selenium tests against AJAX applications

Web applications today are being designed in such a way that they appear the same as desktop applications. Web developers are accomplishing this by using AJAX within their web applications. **AJAX (Asynchronous JavaScript And XML)** relies on JavaScript creating asynchronous calls and then returning XML with the data that the user or application requires to carry on. AJAX does not rely on XML anymore, as more and more people move over to **JSON (JavaScript Object Notation)**, which is more lightweight in the way it transfers the data. It does not rely on the extra overhead of opening and closing tags that is needed to create valid XML.

**Working on pages with AJAX**

In our example, we will click on a link and then assert that some text is visible on the screen:

1. Start up Selenium IDE and make sure that the record button is pressed.
3. Click on the text that says Click this link to load a page with AJAX.
4. Verify the text that appears on your screen. Your test should look like the following screenshot:
5. Run the test that you have created. When it has finished running, it should look like the following screenshot:

![Selenium IDE screen capture]

Have a look at the page that you are working against. Can you see the text that the test is expecting? You should see it, so why has this test failed? The test has failed because when the test reached that point, the element containing the text was not loaded into the DOM. This is because it was being requested and rendered from the web server into the browser.

To remedy this issue, we will need to add a new command to our test so that our tests pass in the future:

1. Right-click on the step that failed so that the Selenium IDE context menu appears.
2. Click on Insert New Command.
3. In the **Command** select box, type `waitForElementPresent` or select it from the drop-down menu.

4. In the **Target** box, add the target that is used in the `verifyText` command.

5. Run the test again and it should pass this time:

![Screenshot of Selenium IDE](image)

Selenium does not implicitly wait for the item that it needs to interact with, so it is seen as good practice to wait for the item you need to work with and then interact with it. The `waitFor` commands will timeout after 30 seconds by default, but if you need it to wait longer, you can specify the tests by using the `setTimeout` command. This will set the timeout value that the tests will use in future commands.

If need be, you can change the default wait if you go to Options | Options and then on the **General** tab, and under **Default timeout value of recorded command in milliseconds** \( (30s = 30,000ms) \) change it to what you want. Remember, 1,000 milliseconds = 1 second.
Working with AJAX applications

As more and more applications try to act like desktop applications, we need to be able to handle synchronization steps between our test and our application. In this section, we will see how to handle AJAX and what to synchronize:

2. Click on the load text to the page button.
3. Wait for the text I have been added with a timeout. Your test will look like the following screenshot:

   ![Screenshot of Selenium IDE with test steps]

In the previous examples, we waited for an element to appear on the page; there are a number of different commands that we can use to wait. Also, remember that we can take advantage of waiting for something not to be on the page, for example, waitForElementNotPresent. This can be just as effective as waiting for it to be there.
The following commands make up the `waitFor` set of commands, but this is not an exhaustive list:

<table>
<thead>
<tr>
<th>Selenium Command</th>
<th>Command Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>waitForAlertNotPresent</code></td>
<td>This waits for an alert to disappear from the page.</td>
</tr>
<tr>
<td><code>waitForAlertPresent</code></td>
<td>This waits for an alert to appear on the page.</td>
</tr>
<tr>
<td><code>waitForElementPresent</code></td>
<td>This waits for an expected element to appear on the page.</td>
</tr>
<tr>
<td><code>waitForElementNotPresent</code></td>
<td>This waits for an expected element to disappear from the page.</td>
</tr>
<tr>
<td><code>waitForTextPresent</code></td>
<td>This waits for expected text and its corresponding HTML tags to appear on the page.</td>
</tr>
<tr>
<td><code>waitForTextNotPresent</code></td>
<td>This waits for expected text and its corresponding HTML tags to disappear from the page.</td>
</tr>
<tr>
<td><code>waitForPageToLoad</code></td>
<td>This waits for all elements to appear on the expected page.</td>
</tr>
<tr>
<td><code>waitForFrameToLoad</code></td>
<td>This waits for an expected frame and its corresponding HTML tags to appear on the page.</td>
</tr>
</tbody>
</table>

A number of these commands are run implicitly when other commands are being run. An example of this is the `clickAndWait` command. This will fire off a `click` command and then fire off a `waitForPageToLoad` command. Another example is the `open` command, which only completes when the page has fully loaded.

If you are feeling confident, then it’s a good time to try different `waitFor` command techniques.

**Storing information from the page in the test**

Sometimes, there is a need to store elements that are on the page to be used later in a test. It could be that your test needs to pick a date that is on the page and use it later so that you do not need to hardcode values into your test.
Once the element has been stored, you will be able to use it again by requesting it from a JavaScript dictionary that Selenium keeps track of. To use the variable, it will take one of the following two formats: it can look like `${variableName}` or `storedVars['variableName']`. I prefer the `storedVars` format as it follows the same format as within Selenium internals. To see how this works, let's work through the following example:

1. Open up Selenium IDE and switch off the record button.
3. Right-click on the text **Assert that this text is on the page** and go to the `storeText` command in the context menu and click on it.
4. A dialog will appear as shown in the following screenshot. Enter the name of a variable that you want to use. I have used `textOnThePage` as the name of my variable.

![Dialog for storing text](image)

5. Click on the row below the `storeText` command in Selenium IDE.
6. Type `type` into the **Command** textbox.
7. Type `storeinput` into the **Target** box.
8. Type `${textOnThePage}` into the **Value** box.
9. Run the test. It should look like the following screenshot:

![Selenium IDE 2.8.0 screenshot](image)

Once your test has completed running, you will see that it has placed **Assert that this text is on the page** into the textbox.

**Debugging tests**

We have successfully created a number of tests and have seen how we can work against AJAX applications, but unfortunately, creating tests that run perfectly the first time can be difficult. Sometimes, as a test automator, you will need to debug your tests to see what is wrong.

To work through this part of the chapter, you will need to have a test open in Selenium IDE.
Getting Started with Selenium IDE

These two steps are quite useful when your tests are not running and you want to execute a specific command. They are:

1. Highlight a command.
2. Press the X key. This will make the command execute in Selenium IDE.

When a test is running, you can press the pause button to pause the test after the step that is currently being run. Once the test has been paused, the step button is no longer disabled and you can press it to step through the test as if you were stepping through an application.

If you are having issues with elements on the page, you can type in their location and then click on the Find button. This will surround the element that you are looking for with a green border that flashes for a few seconds. It should look like the following screenshot:

The echo command is also a good way to write something from your test to the log. This is equivalent to the Console.log code in JavaScript, for example, echo | ${textOnThePage}, as shown in the following screenshot:
Also, remember that if you are trying to debug a test script that you have created with Selenium IDE, you can set breakpoints in your test. You simply right-click on the line and select breakpoint from the list. It will be similar to the following screenshot:

You can also use the keyboard shortcut of B to allow you to do it quicker.
Creating test suites

We managed to create a number of tests using Selenium IDE and have managed to run them successfully. The next thing to have a look at is how to create a test suite, so that we can open the test suite and then have it run a number of tests that we have created. If you have Selenium IDE open from the last steps, click on the **File** menu:

1. Click **New Test Case**.
2. You will see that Selenium IDE has opened a new area on the left of the IDE as shown in the following screenshot:

![Screenshot of Selenium IDE test case](image)

You can do this as many times as you want, and when the **Play entire test suite** button is clicked, it will run all the tests in the test suite. It will log all the passes and failures at the bottom of the **Test Case** box.

To save this, click on the **File** menu and then click **Save Test Suite** and save the test suite file to a place where you can get to it again. One thing to note is that saving a test suite does not save the test case. Make sure that you save the test case every time you make a change and not just the test suite.
To change the name of the test case to something a lot more meaningful, you can do this by right-clicking on the test and clicking on the **Properties** item in the context menu:

![Image of Properties dialog]

You can now add meaningful names to your tests and they will appear in Selenium IDE instead of falling back to their filenames.

We have managed to create our first test suite. This can be used to group tests together to be used later. If your tests have been saved, you can update the test suite properties to give the tests a name that is easier to read.

### Parameterization

Parameterization is a key feature in automation testing. It is possible to give test data (parameter) as input in Selenium IDE as it is with any other automation tool. Selenium IDE requires configuration before proceeding for parameterization, which is explained in the following section.

After configuring Selenium IDE for parameterization, in the following example, we will search a text on Google and verify the text appears on the page. The `storeEval` command will read the search data from the `google_search.js` file and save it in the `searchDataValue` variable. With this variable, data types on the Google search page, verify **Download Selenium IDE** text using the `assertTextPresent` command. In this way, you can achieve data parameterization or data-driven testing by passing external data to a variable in Selenium IDE.
Selenium IDE Configuration

You need to create JavaScript file as the data source file. These are the steps to create JavaScript file and configure Selenium IDE:

1. In a Notepad, create input data as follows:
   ```javascript
   searchData = 'Selenium IDE'
   ```

2. Save it as a file, say for example, `google_search.js`.

3. Open Selenium IDE and choose the **Option | Options...** menu.

4. Under Selenium IDE Extensions, browse the saved `google_search.js` file. After attaching the data file, your Selenium IDE option window will look like the following screenshot. Then, click **OK** and restart the Selenium IDE:

5. Now, record the script to perform Google search and verify the page element and update the commands as shown in the following screenshot. Then, execute the test. You can also use the **WaitForElementPresent** command till the page element is loaded and verify instead of the **pause** command.
In this example, we are searching a text on the Google page and verifying the text appearing on the page. The `storeEval` command will read the search data from the `google_search.js` file and store it in a `searchDataValue` variable and verify the `Download Selenium IDE` text using an `assertTextPresent` command. In this way, you can achieve data parameterization or data-driven testing in Selenium IDE.

**Generating source code from Selenium IDE**

The Selenium IDE formatters plugin enables you to export the recorded steps. Selenium IDE can export the recorded steps or user actions to different formats. The Selenium team provides bindings for several programming languages. You can write a script in one of these programming languages and use the provided libraries to remotely control a browser. Selenium IDE formatters convert the recorded actions into source code for a specific programming language.
You need to set enable experimental features on Selenium IDE options to view the formatters. Check out the following screenshot for formatters:
Record user actions on Selenium IDE, and to view specific programming language formatters, select the **Selenium IDE | Options | Format** as shown in the following screenshot. A confirmation dialog appears, click on **OK**:

![Selenium IDE screenshot](image-url)
Getting Started with Selenium IDE.

Ensure that the code is displayed under the Selenium IDE Source tab as seen in the screenshot that follows. You can also see the Java/Junit4/WebDriver programming language formatter. Selenium IDE used the type command to identify text typing into the textbox. In the source code, the sendKeys command is used in order to type text into the textbox using the Selenium Java/Junit4/WebDriver programming language. You can generate a formatter of any of the specific programming languages. After viewing the source code, you can switch back from a specific formatter to HTML. If a formatter is not HTML, then recording options will not work and the Table tab will not be active:

```java
import org.openqa.selenium.FirefoxDriver;
import org.openqa.selenium.WebDriver;
import static org.junit.Assert.assertEquals;
import static org.openqa.selenium.By.id;
import static org.openqa.selenium.By.name;
import static org.openqa.selenium.support.ui.ExpectedConditions;
import static org.openqa.selenium.support.ui.WebDriverWait;

public class Parameterization {
    private WebDriver driver;
    private String baseUrl;
    private boolean acceptNextAlert = true;
    private StringBuffer verificationErrors = new StringBuffer();

    @Before
    public void setUp() throws Exception {
        driver = new FirefoxDriver();
        baseUrl = "https://www.google.com/";
        driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
    }

    @Test
    public void testParameterization() throws Exception {
        driver.get(baseUrl + "/");
        // Expected: Unsupported command [getEval]
        driver.findElement(By.name("q")).clear();
        driver.findElement(By.name("q")).sendKeys(searchDataValue);
        driver.findElement(By.id("gbqsb")).click();
        // Warning: assertTextPresent may require manual changes
        assertEquals(driver.findElement(By.cssSelector("BODY")).getText(),
```

In the preceding recorded action, we searched a text on Google and verified the text appears on the page. Selenium IDE exports these actions to view a specific programming language. This is one of the useful features of Selenium IDE; it provides an option to set the clipboard format. You can copy commands from the Table tab and paste in the format of that specific programming language.

Saving tests
Saving tests is done in the same manner as saving a test suite. Click on the File menu and then click Save Test Case. This will give you a Save dialog box; save this to a place where you can get to it later. When you save your tests and your test suite, Selenium IDE will try to keep the relationships between the folders in step when saving the tests and the test suites.

What you cannot record
We have seen our tests work really well by recording them and then playing them back. Unfortunately, there are a number of things that Selenium cannot do. Since Selenium was developed in JavaScript, it tries to synthesize what the user does with JavaScript events. This means that it is bound by the same rules that JavaScript has in any browsers by operating within the sandbox.

- Silverlight and Flex/Flash applications, at the time of writing, cannot be recorded with Selenium IDE. Both these technologies operate in their own sandbox and do not operate with the DOM to do their work. HTML5, at the time of writing, is not fully supported with Selenium IDE. A good example of this is elements that have the contentEditable=true attribute. If you want to see this, you can use the type command to type something into the html5div element. The test will tell you that it has completed the command, but the UI will not have changed, as shown in the following screenshot:
Getting Started with Selenium IDE

- Selenium IDE does not work with Canvas elements on the page either, so you will not be able to make your tests move items around on a page.
- Selenium cannot do file uploads. This is due to the JavaScript sandbox not allowing JavaScript to interact with `<input type=file>` elements on a page. While you might be able to send the text to the box, it will not always do what you expect, so I would recommend not doing it.

We will be able to automate a number of these elements with Selenium WebDriver in later chapters of this book.

Summary

We learned a lot in this chapter about Selenium IDE, learning how to create your first test using the record and replay functions and we now understand some of the basic concepts such as moving between multiple windows that can appear in a test, and saving our tests for future use.

Specifically, we covered the following topics:

- **How to install Selenium IDE**: We started by downloading Selenium IDE from [http://seleniumhq.org](http://seleniumhq.org).
- **What Selenium IDE is made up of**: The breakup of Selenium IDE allowed us to see what makes up Selenium IDE. It allowed us to understand the different parts that make up a command that will be executed in a test as well as its basic format. We had a look at how to load Selenium IDE and how to get started with recording tests. We saw that a Selenium IDE command is made up of three sections: the command, the target, and the value that might be used.
- **Recording and replaying tests**: We used Selenium IDE to record a workflow that a user will need in their tests. We also had a look at verifying and asserting that elements are on the page and that the text we are expecting is also on the page.
- **How to add comments to tests**: In this section of the chapter, we saw how to add comments to the tests so that they are more maintainable.
- **Working with multiple windows**: In this section, we saw how applications today can have pop-up windows that tests need to be able to move between.
Chapter 1

- **Working with AJAX applications**: AJAX applications do not have the items needed for the tests when the tests get to commands. To get around this, we had a look at adding `waitFor` commands to the tests. This is due to the fact that Selenium does not implicitly wait for elements to appear on the page.

- **Storing information in variables**: There is always something that is on the page that needs to be used later, but unfortunately, you will not know what the value is before the test runs. This section showed us how we can record items into a variable and use it later in a test. This can be something that has happened on a page and needs to be checked that it is still there on later pages.

- **Debugging tests**: Creating tests does not always go according to the plan, so in this section, we saw some of the different ways to debug your tests.

- **Parameterization**: Parameterizing the data in Selenium IDE.

- **Saving test suites**: Finally, we saw how we can save tests for future use and how we can save them into different groups by saving them into test suites.

We also discussed what cannot be tested using Selenium IDE. We saw that Silverlight and Flex/Flash applications cannot be tested, and that when working with a number of HTML5 elements, the tests say that they have completed the tasks even though the UI has not changed. In later chapters, we will discuss different mechanisms that we can use within our tests that might be useful against HTML5 elements on the page.

Now that we've learnt about Selenium IDE, we're ready to look at all the different techniques we can use to find elements on the page.

**Self-test questions**

1. What is the main language that drives Selenium IDE?
   1. Ruby.
   2. Python.

2. Selenium IDE works on Internet Explorer.
   1. True.
   2. False.
3. Selenium verifies items on the page when it is recording steps.
   1. True.
   2. False.

4. What is the difference between verify and assert?

5. If you wanted to validate that a button had appeared on a page, which two commands would be the best to use?
   1. verifyTextPresent / assertTextPresent.
   2. verifyElementPresent / assertElementPresent.
   3. verifyAlertPresent / assertAlertPresent.
   4. verifyAlert / assertAlert.

6. If an element got added after the page loaded, what command would you use to make sure the test passed in the future?
   1. waitForElementPresent.
   2. pause.
   3. assertElementPresent.

7. How do we run all the tests in a test suite?
Where to buy this book


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.