I. Selenium WebDriver
- Introducing WebDriver
- How Does WebDriver 'Drive' the Browser Compared to Selenium-RC?
- WebDriver and the Selenium-Server
- Introducing the Selenium-WebDriver API by Example
- Selenium-WebDriver API Commands and Operations
    - Fetching a Page
    - Locating UI Elements (WebElements)
        - By ID
        - By Tag Name
        - By Name
        - By Link Text
        - By Partial Link Text
        - By CSS
        - By XPath
    - Using JavaScript
    - Getting text values
    - User Input - Filling In Forms
    - Moving Between Windows and Frames
    - Popup Dialogs
    - Navigation: History and Location
    - Cookies
    - Changing the User Agent
    - Drag And Drop

II. Using the Page Object Model 145
- Introduction
- Using the PageFactory class for exposing elements from a page
- Using the PageFactory class for exposing an operation on a page

III. Driver Specifics and Tradeoffs
- Selenium-WebDriver's Drivers
    - HtmlUnit Driver
    - Firefox Driver
    - Internet Explorer Driver
    - ChromeDriver

IV. Running Standalone Selenium Server for use with RemoteDrivers

V. WebDriver: Advanced Usage
- Explicit and Implicit Waits
- RemoteWebDriver
- AdvancedUserInteractions
- Parallelizing Your Test Runs

VI. JavaScript and Selenium JavaScriptExecutor
- What is JavaScript?
- Evaluating Xpaths
- Finding Element using JavaScript and JavaScriptExecutor

- Generate Alert Pop Window
- Click Action
- Refresh Browser
- Get InnerText of a Webpage
- Get Title of a WebPage
- Scroll Page

VII. Selenium-Grid
- What is Selenium-Grid?
- When to Use It
- Selenium-Grid 2.0
- How Selenium-Grid Works–With a Hub and Nodes
- Starting Selenium-Grid
- Hub Node Configuration

VIII. TestNG
- Introduction
- Annotations
- testng.xml
- Running TestNG
- Test methods, Test classes and Test groups, Parameters
- Parallelism and time-outs

IX. What is Apache Ant
- Benefit of Ant build
- Understanding Build.xml

X. Maven & Jenkins with Selenium
- What is Jenkins?
- Important Features of Jenkins
- Why Jenkins and Selenium?
- Why Maven & Jenkins

XI. Introduction Java
- Java Basics
- What Is a Collections Framework?
- Benefits of the Java Collections Framework
- Collection Interfaces
- ArrayList
- Summary

Selenium WebDriver

Introducing WebDriver

The primary new feature in Selenium 2.0 is the integration of the WebDriver API. WebDriver is designed to provide a simpler, more concise programming interface in addition to addressing some limitations in the Selenium-RC API. Selenium-WebDriver was developed to better support dynamic web pages where elements of a page may change without the page itself being reloaded. WebDriver's goal is to supply a well-designed object-oriented API that provides improved support for modern advanced web-app testing problems.

How Does WebDriver 'Drive' the Browser Compared to Selenium-RC?

Selenium-WebDriver makes direct calls to the browser using each browser's native support for automation. How these direct calls are made, and the features they support depends on the browser you are using.

For those familiar with Selenium-RC, this is quite different from what you are used to. Selenium-RC worked the same way for each supported browser. It 'injected' javascript functions into the browser when the browser was loaded and then used its javascript to drive the AUT within the browser. WebDriver does not use this technique. Again, it drives the browser directly using the browser's built in support for automation.

WebDriver and the Selenium-Server

You may, or may not, need the Selenium Server, depending on how you intend to use Selenium-WebDriver. If your browser and tests will all run on the same machine, and your tests only use the WebDriver API, then you do not need to run the Selenium-Server; WebDriver will run the browser directly.

There are some reasons though to use the Selenium-Server with Selenium-WebDriver.

- You are using Selenium-Grid to distribute your tests over multiple machines or virtual machines (VMs).
- You want to connect to a remote machine that has a particular browser version that is not on your current machine.
- You are not using the Java bindings (i.e. Python, C#, or Ruby) and would like to use HtmlUnit Driver

Introducing the Selenium-WebDriver API by Example

WebDriver is a tool for automating web application testing, and in particular to verify that they work as expected. It aims to provide a friendly API that's easy to explore and understand, easier to use than the Selenium-RC (1.0) API, which will help to make your tests easier to read and maintain. It's not tied to any particular test framework, so it can be used equally well in a unit testing project or from a plain old "main" method. This section introduces WebDriver's API and helps get you started becoming familiar with it.

Once your project is set up, you can see that WebDriver acts just as any normal library: it is entirely self-contained, and you usually don't need to remember to start any additional processes or run any installers before using it, as opposed to the proxy server with Selenium-RC.

You're now ready to write some code. An easy way to get started is this example, which searches for the term "Cheese" on Google and then outputs the result page's title to the console.

```
//java
packageorg.openqa.selenium.example;
importorg.openqa.selenium.By;
importorg.openqa.selenium.WebDriver;
importorg.openqa.selenium.WebElement;
importorg.openqa.selenium.firefox.FirefoxDriver;
importorg.openqa.selenium.support.ui.ExpectedCondition;
importorg.openqa.selenium.support.ui.WebDriverWait;

public class Selenium2Example  {
public static void main(String[] args) {
    // Create a new instance of the Firefox driver
    // Notice that the remainder of the code relies on the interface,
    // not the implementation.
    WebDriver driver = new FirefoxDriver();

    // And now use this to visit Google
driver.get("http://www.google.com");
    // Alternatively the same thing can be done like this
    // driver.navigate().to("http://www.google.com");

    // Find the text input element by its name
WebElement element = driver.findElement(By.name("q"));

    // Enter something to search for
element.sendKeys("Cheese!");

    // Now submit the form. WebDriver will find the form for us from the element
element.submit();

    // Check the title of the page
System.out.println("Page title is: " + driver.getTitle());
```

```
        // Google's search is rendered dynamically with JavaScript.
        // Wait for the page to load, timeout after 10 seconds
        (newWebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
public Boolean apply(WebDriver d) {
returnd.getTitle().toLowerCase().startsWith("cheese!");
          }
        });

        // Should see: "cheese! - Google Search"
System.out.println("Page title is: " + driver.getTitle());

        //Close the browser
driver.quit();
   }
}
```

Refer document https://www.seleniumeasy.com/selenium-webdriver-tutorials for more knowledge.

Selenium-WebDriver API Commands and Operations

- **Fetching a Page**

The first thing you're likely to want to do with WebDriver is navigate to a page. The normal way to do this is by calling "get":

```
//java

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

Dependent on several factors, including the OS/Browser combination, WebDriver may or may not wait for the page to load. In some circumstances, WebDriver may return control before the page has finished, or even started, loading. To ensure robustness, you need to wait for the element(s) to exist in the page using Explicit and Implicit Waits.

- **Locating UI Elements (WebElements)**

Locating elements in WebDriver can be done on the WebDriver instance itself or on a WebElement. Each of the language bindings exposes a "Find Element" and "Find Elements" method.

The former returns a WebElement object matching the query, and throws an exception if such an element cannot be found.

The latter returns a list of WebElements, possibly empty if no DOM elements match the query.

The "Find" methods take a locator or query object called "By".

"By" strategies are listed below.

I. By ID

This is the most efficient and preferred way to locate an element. Common pitfalls that UI developers make is having non-unique id's on a page or auto-generating the id, both should be avoided. A class on an html element is more appropriate than an auto-generated id.

Example of how to find an element that looks like this:

```
<div id="coolestWidgetEvah">...</div>

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

II. By Class Name

"Class" in this case refers to the attribute on the DOM element. Often in practical use there are many DOM elements with the same class name, thus finding multiple elements becomes the more practical option over finding the first element.

Example of how to find an element that looks like this:

```
<div class="cheese"><span>Cheddar</span></div><div
class="cheese"><span>Gouda</span></div>

List<WebElement> cheeses = driver.findElements(By.className("cheese"));
```

III. By Tag Name

The DOM Tag Name of the element.

Example of how to find an element that looks like this:

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

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

IV. By Name

Find the input element with matching name attribute.

Example of how to find an element that looks like this:

```
<input name="cheese" type="text"/>

WebElement cheese = driver.findElement(By.name("cheese"));
```

V. By Link Text

Find the link element with matching visible text.

Example of how to find an element that looks like this:

```
<ahref="http://www.google.com/search?q=cheese">cheese</a>>

WebElement cheese = driver.findElement(By.linkText("cheese"));
```

VI. By Partial Link Text

Find the link element with partial matching visible text.

Example of how to find an element that looks like this:

```
<a href="http://www.google.com/search?q=cheese">search for cheese</a>>

WebElement cheese = driver.findElement(By.partialLinkText("cheese"));
```

VII. By CSS

Cascading Style Sheets (CSS) Selector is combination of an element selector and a selector value which identifies the web element within a web page. The composite of element selector and selector value is known as Selector Pattern.

Like the name implies it is a locator strategy by css. Native browser support is used by default, so please refer to w3c css selectors ("https://www.w3.org/TR/CSS/#selectors") for a list of generally available css selectors.

Example of to find the cheese below:

```
<div id="food">
        <span class="dairy">milk</span>
        <span class="dairy aged">cheese</span>
</div>

WebElement cheese = driver.findElement(By.cssSelector("#food span.dairy.aged"));
```

VIII. By XPath

At a high level, WebDriver uses a browser's native XPath capabilities wherever possible. On those browsers that don't have native XPath support, we have provided our own implementation. This can lead to some unexpected behavior unless you are aware of the differences in the various XPath engines.

Driver	Tag and Attribute Name	Attribute Values	Native XPath Support
HtmlUnit Driver	Lower-cased	As they appear in the HTML	Yes
Internet Explorer Driver	Lower-cased	As they appear in the HTML	No
Firefox Driver	Case insensitive	As they appear in the HTML	Yes

This is a little abstract, so for the following piece of HTML:

```
<input type="text" name="example" />
<INPUT type="text" name="other" />
```

```
List<WebElement> inputs = driver.findElements(By.xpath("//input"));
```

The following number of matches will be found

XPath expression	HtmlUnit Driver	Firefox Driver	Internet Explorer Driver
//input	1 ("example")	2	2
//INPUT	0	2	0

Sometimes HTML elements do not need attributes to be explicitly declared because they will default to known values. For example, the "input" tag does not require the "type" attribute because it defaults to "text". The rule of thumb when using xpath in WebDriver is that you should not expect to be able to match against these implicit attributes.

- **Using JavaScript**

You can execute arbitrary javascript to find an element and as long as you return a DOM Element, it will be automatically converted to a WebElement object.

Simple example on a page that has jQuery loaded:

WebElement element = (WebElement) ((JavascriptExecutor)driver).executeScript("return $('.cheese')[0]");

- **Getting text values**

People often wish to retrieve the innerText value contained within an element. This returns a single string value. Note that this will only return the visible text displayed on the page.

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

element.getText();
```

- **User Input - Filling In Forms**

Dealing with SELECT tags isn't too bad:

```
WebElement select = driver.findElement(By.tagName("select"));
List<WebElement>allOptions = select.findElements(By.tagName("option"));
for (WebElement option : allOptions) {
        System.out.println(String.format("Value is: %s", option.getAttribute("value")));
        option.click();
}
```

This will find the first "SELECT" element on the page, and cycle through each of its OPTIONs in turn, printing out their values, and selecting each in turn. As you will notice, this isn't the most efficient way of dealing with SELECT elements. WebDriver's support classes include one called "Select", which provides useful methods for interacting with these.

```
Select select = new Select(driver.findElement(By.tagName("select")));
select.deselectAll();
select.selectByVisibleText("Edam");
```

This will deselect all OPTIONs from the first SELECT on the page, and then select the OPTION with the displayed text of “Edam”.

Once you’ve finished filling out the form, you probably want to submit it. One way to do this would be to find the “submit” button and click it:

```
driver.findElement(By.id("submit")).click();
```

Alternatively, WebDriver has the convenience method “submit” on every element. If you call this on an element within a form, WebDriver will walk up the DOM until it finds the enclosing form and then calls submit on that. If the element isn’t in a form, then theNoSuchElementException will be thrown:

```
element.submit();
```

- **Moving Between Windows and Frames**

Some web applications have many frames or multiple windows. WebDriver supports moving between named windows using the “switchTo” method:

```
driver.switchTo().window("windowName");
```

All calls to driver will now be interpreted as being directed to the particular window. But how do you know the window’s name? Take a look at the javascript or link that opened it:

Click here to open a new window

Alternatively, you can pass a “window handle” to the “switchTo().window()” method. Knowing this, it’s possible to iterate over every open window like so:

```
for (String handle : driver.getWindowHandles()) {
driver.switchTo().window(handle);
}
```

You can also switch from frame to frame (or into iframes):

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

- **Popup Dialogs**

Starting with Selenium 2.0 beta 1, there is built in support for handling popup dialog boxes. After you’ve triggered an action that opens a popup, you can access the alert with the following:

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

This will return the currently open alert object. With this object you can now accept, dismiss, read its contents or even type into a prompt. This interface works equally well on alerts, confirms, and prompts. Refer to the JavaDocs(https://seleniumhq.github.io/selenium/docs/api/java/index.html) for more information.

- **Navigation: History and Location**

Earlier, we covered navigating to a page using the "get" command

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

As you've seen, WebDriver has a number of smaller, task-focused interfaces, and navigation is a useful task. Because loading a page is such a fundamental requirement, the method to do this lives on the main WebDriver interface, but it's simply a synonym to:

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

To reiterate: "navigate().to()" and "get()" do exactly the same thing. One's just a lot easier to type than the other!

The "navigate" interface also exposes the ability to move backwards and forwards in your browser's history:

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

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

Please be aware that this functionality depends entirely on the underlying browser. It's just possible that something unexpected may happen when you call these methods if you're used to the behavior of one browser over another.

- **Cookies**

Before we leave these next steps, you may be interested in understanding how to use cookies. First of all, you need to be on the domain that the cookie will be valid for. If you are trying to preset cookies before you start interacting with a site and your homepage is large / takes a while to load an alternative is to find a smaller page on the site (typically the 404 page is small, e.g. http://example.com/some404page).

```
// Go to the correct domain
driver.get("http://www.example.com");
// Now set the cookie. This one's valid for the entire domain
Cookie cookie = new Cookie("key", "value");
driver.manage().addCookie(cookie);
// And now output all the available cookies for the current URL
```

```
Set<Cookie>allCookies = driver.manage().getCookies();
for (Cookie loadedCookie : allCookies) {
        System.out.println(String.format("%s -> %s", loadedCookie.getName(),
        loadedCookie.getValue()));
}
// You can delete cookies in 3 ways
// By name
driver.manage().deleteCookieNamed("CookieName");
// By Cookie
driver.manage().deleteCookie(loadedCookie);
// Or all of them
driver.manage().deleteAllCookies();
```

- **Changing the User Agent**

An example for the same in Firefox Driver:

```
FirefoxProfile profile = new FirefoxProfile();
profile.addAdditionalPreference("general.useragent.override", "some UA string");
WebDriver driver = new FirefoxDriver(profile);
```

- **Drag And Drop**

Here's an example of using the Actions class to perform a drag and drop. Native events are required to be enabled.

```
WebElement element = driver.findElement(By.name("source"));
WebElement target = driver.findElement(By.name("target"));
(new Actions(driver)).dragAndDrop(element, target).perform();
```

Using the Page Object Model

Introduction

Developing a maintainable automation code is one of the keys to a successful test-automation project. Test-automation code needs to be treated as production code and similar standards and patterns should to be applied while developing this code.

While developing Selenium WebDriver tests, we can use the Page Object model pattern. This pattern helps in enhancing the tests, making them highly maintainable, reducing the code duplication, building a layer of abstraction, and hiding the inner implementation from tests.

Using the Page Object Model:

By applying object-oriented development principles, we can develop a class that serves as an interface to a web page in the application, modeling its properties and behavior. This helps in creating a layer of separation between the test code and code specific to the page, by hiding the technical implementation such as locators used to identify elements on the page, layout, and so on. The Page Object design pattern provides tests for an interface where a test can operate on that page in a manner similar to the user accessing the page, but by hiding its internals. For example, if we build a Page Object test for a login page, then it will provide a method to log in, which will accept the username and password and take the user to the home page of the application. The test need not worry about how input controls are used for the login page, their locator details, and so on.

Tests should use objects of a page at a high level, where any change Using the PageFactory class for exposing elements from a page

POM is aIt is fundamental Unit of Work in Maven and It is an XML file.It contains information about the project and various configuration details used by Maven to build the project(s).

Using the PageFactory class for exposing elements from a page

For implementing the Page Object model in tests, we need to map and create a Page Object class for each page being tested. For example, to test the BMI Calculator application, a BMI Calculator page class will be defined, which will expose the internals of the BMI Calculator page to the test, as shown in following diagram. This is done by using the PageFactory class of Selenium WebDriver API.

Before exposing the elements of a page, we need to do the following:

- Identify locators that will be needed to locate the elements uniquely
- Define the structure of the package and classes for objects of the page

How to do it...

Let's implement a Page Object test for the BMI Calculator page using the PageFactory class with the following steps:

1. Define and create a package for all the page's objects from the application for logical grouping. For this example, seleniumcookbook.tests.pageobjects is created to define all the page's objects.

2. Create a new Java class file. Give the name of the page we will be testing from the application to this class. For example, we will be creating a page object for the BMI Calculator application, so the class name could be BmiCalcPage. This is a single page application. The Java class file would have the following code:

```
packageseleniumcookbook.tests.pageobjects;
importorg.openqa.selenium.WebDriver;
importorg.openqa.selenium.WebElement;
importorg.openqa.selenium.support.PageFactory;
public class BmiCalcPage {
}
```

3. Define elements from the BMI Calculator page as instance variables in the BmiCalcPage class created in step 1. Use the name or id attributes to name these variables as follows:

```
publicWebElementheightCMS;
publicWebElementweightKg;
publicWebElement Calculate;
publicWebElementbmi;
publicWebElementbmi_category;
```

4. Add a constructor to the BmiCalcPage class, which will call the PageFactory. initElements() method to initialize the elements in the class. In other words, map the elements to the variables in the BmiCalcPage class as follows:

```
publicBmiCalcPage(WebDriver driver) {
PageFactory.initElements(driver, this);
}
```

5. Finally, create a test which will use the BmiCalcPage class for testing the BMI Calculator page as follows:

```
packageseleniumcookbook.tests;
importorg.openqa.selenium.WebDriver;
importorg.openqa.selenium.chrome.ChromeDriver;
importorg.junit.Test;
import static org.junit.Assert.*;
importseleniumcookbook.tests.pageobjects.*;
public class BmiCalculatorTests {
@Test
```

```
public void testBmiCalculation()
{
//Open Chrome Browser
//and navigate to BMI Calculator Page
WebDriver driver = new ChromeDriver();
driver.get("http://dl.dropbox.com/u/55228056/bmicalculator.html");
//Create instance of BmiCalcPage and pass the driver
BmiCalcPagebmiCalcPage = new BmiCalcPage(driver);
//Enter Height & Weight
bmiCalcPage.heightCMS.sendKeys("181");
bmiCalcPage.weightKg.sendKeys("80");
//Click on Calculate button
bmiCalcPage.Calculate.click();
//Verify Bmi&Bmi Category values
assertEquals("24.4", bmiCalcPage.bmi.getAttribute("value"));
assertEquals("Normal", bmiCalcPage.bmi_category.
getAttribute("value"));
//Close the Browser
driver.close();
}
}
```

Using the Page Object model and the PageFactory class, the BMI Calculator page's elements are exposed through the BmiCalcPage class to the test instead of the test directly accessing the internals of the page.

When we initialize the page's object using the PageFactory class in the BmiCalcPage class, the PageFactory class searches for the elements on the page with the name or id attributes matching the name of the WebElement object declared in the BmiCalcPage class, as follows:

```
publicBmiCalcPage(WebDriver driver) {
PageFactory.initElements(driver, this);
}
```

The initElements() method takes the driver object created in the test and initializes the elements declared in the BmiCalcPage class. We can then directly call the methods on these elements as follows:

```
bmiCalcPage.heightCMS.sendKeys("181");
bmiCalcPage.weightKg.sendKeys("80");
bmiCalcPage.Calculate.click();
```

FindBy annotations

Finding elements using the name or id attributes may not always work and we might need to use advanced locator strategies such as XPath or CSS selectors. Using the FindBy annotation, we can locate the elements within the PageFactory class as follows:

```
@FindBy(id = "heightCMS")
publicWebElementheightField;
```

We declared a public member for the height element and used the @FindBy annotation, specifying the id as a locator for finding this element on the page.

CacheLookUp Attribute

One downside to using the @FindBy annotation is that every time we call a method on the WebElement object, the driver will go and find it on the current page again. This is useful in applications where elements are dynamically loaded or AJAX-heavy applications. However, in applications where we know that the element is always going to be there and stay the same without any change, it would be handy if we could cache the element once we find it. We can use the @CacheLookUp annotation along with the @FindBy annotation as follows in order to do this:

```
@FindBy(id = "heightCMS")
@CacheLookup
publicWebElementheightField;
```

This tells the PageFactory.initElements() method to cache the element once it's located. Tests work faster with cached elements when these elements are used repeatedly.

Using the PageFactory class for exposing an operation on a page

In the previous recipe, we created the BmiCalcPage class, which provides elements from the BMI Calculator page to the test. Along with elements, we define operations or behaviors on a page. In the BMI Calculator application, we are calculating the BMI by entering height and weight values. We can create an operation named calculateBmi and call it directly in a test, instead of calling individual elements and operations.

In this recipe, let's refine the BmiCalcPage class and instead of elements, provide the operations that are supported on the page and some common properties. We will also move the WebDriver instance of the test to the BmiCalcPage class to make the test generic.

Let's modify the BmiCalcPage class created in the previous recipe and refactor it a bit to provide operations and properties to the test through the following steps:

1) Make the page's elements private in the BmiCalPage class for better encapsulation. Also add an instance variable of the WebDriver class and a string variable for the URL of the page as follows:

```
packageseleniumcookbook.tests.pageobjects;
importorg.openqa.selenium.WebDriver;
importorg.openqa.selenium.chrome.ChromeDriver;
importorg.openqa.selenium.WebElement;
importorg.openqa.selenium.support.PageFactory;
public class BmiCalcPage {
privateWebElementheightCMS;
privateWebElementweightKg;
privateWebElement Calculate;
privateWebElementbmi;
privateWebElementbmi_category;
private WebDriver driver;
private String url = "http://dl.dropbox.com/u/55228056/
bmicalculator.html";
}
```

2. Update the BmiCalcPage class constructor so that it initializes the WebDriver instance as follows:

```
publicBmiCalcPage() {
driver = new ChromeDriver();
PageFactory.initElements(driver, this);
}
```

3. Add the load() and close() methods to the BmiCalcPage class as follows:

```
public void load() {
this.driver.get(url);
}
public void close() {
this.driver.close();
}
```

4. Add the calculateBmi() method to the BmiCalcPage class as follows:

```
public void calculateBmi(String height, String weight) {
heightCMS.sendKeys(height);
weightKg.sendKeys(weight);
Calculate.click();
}
```

5. Add the getBmi() and getBmiCategory() methods to the BmiCalcPage class as follows:

```
public String getBmi() {
returnbmi.getAttribute("value");
}
```

```
public String getBmiCategory() {
returnbmi_category.getAttribute("value");
}
```

6. Finally, create a test that will use these methods defined in the BmiCalcPage class for testing the BMI Calculator page as follows:

```
packageseleniumcookbook.tests;
importorg.junit.Test;
import static org.junit.Assert.*;
importseleniumcookbook.tests.pageobjects.*;
public class BmiCalculatorTests {
@Test
public void testBmiCalculation()
{
//Create an instance of Bmi Calculator Page class
//and provide the driver
BmiCalcPagebmiCalcPage = new BmiCalcPage();
//Open the Bmi Calculator Page
bmiCalcPage.load();
//Calculate the Bmi by supplying Height and Weight values
bmiCalcPage.calculateBmi("181", "80");
//Verify Bmi&Bmi Category values
assertEquals("24.4", bmiCalcPage.getBmi());
assertEquals("Normal", bmiCalcPage.getBmiCategory());
//Close the Bmi Calculator Page
bmiCalcPage.close();
}
}
```

How it works...

In this example, the BmiCalcPage class defines various methods for loading, closing the page, calculation functionality, and providing access to elements as properties to the test. This simplifies the test development by creating a layer of abstraction, hiding the internals of a page, and exposing only operations and fields needed for testing from the page. When any change happens to the structure or behavior of the page, only the BmiCalcPageclass will be refactored while the test will remain intact. In this example, we created the load() method, which navigates the application using the URL as follows:

```
public void load() {
this.driver.get(url);
}
```

Using this approach, we can also expose the elements as properties, which provide specific attributes such as the value attribute instead of exposing elements fully. For example, the getBmi() method provides only the value attribute of the bmi label to the test as follows:

```
public String getBmi() {
returnbmi.getAttribute("value");
}
```

Selenium WebDriver provides a very neat and clean way to implement the Page Object model.

Driver Specifics and Tradeoffs

Selenium-WebDriver's Drivers

WebDriver is the name of the key interface against which tests should be written, but there are several implementations. These include:

- **HtmlUnit Driver**

This is currently the fastest and most lightweight implementation of WebDriver. As the name suggests, this is based on HtmlUnit. HtmlUnit is a java based implementation of a WebBrowser without a GUI. For any language binding (other than java) the Selenium Server is required to use this driver.

Usage

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

Pros

- Fastest implementation of WebDriver
- A pure Java solution and so it is platform independent.
- Supports JavaScript

Cons

- Emulates other browsers' JavaScript behaviour (see below)

- **Firefox Driver**

Controls the Firefox browser using a Firefox plugin. The Firefox Profile that is used is stripped down from what is installed on the machine to only include the Selenium WebDriver.xpi (plugin). A few settings are also changed by default (see the source to see which ones) Firefox Driver is capable of being run and is tested on Windows, Mac, Linux. Currently on versions 3.6, 10, latest - 1, latest

Usage

WebDriver driver = new FirefoxDriver();

Pros

- Runs in a real browser and supports JavaScript
- Faster than the Internet Explorer Driver

Cons

- Slower than the HtmlUnit Driver

Modifying the Firefox Profile

Suppose that you wanted to modify the user agent string (as above), but you've got a tricked out Firefox profile that contains dozens of useful extensions. There are two ways to obtain this profile. Assuming that the profile has been created using Firefox's profile manager (firefox -ProfileManager):

```
ProfilesIniallProfiles = new ProfilesIni();
FirefoxProfile profile = allProfiles.getProfile("WebDriver");
profile.setPreferences("foo.bar", 23);
WebDriver driver = new FirefoxDriver(profile);
```

Alternatively, if the profile isn't already registered with Firefox:

```
File profileDir = new File("path/to/top/level/of/profile");
FirefoxProfile profile = new FirefoxProfile(profileDir);
profile.addAdditionalPreferences(extraPrefs);
WebDriver driver = new FirefoxDriver(profile);
```

- **Internet Explorer Driver**

The InternetExplorerDriver is a standalone server which implements WebDriver's wire protocol. This driver has been tested with IE 7, 8, 9, 10, and 11 on appropriate combinations of Vista, Windows 7, Windows 8, and Windows 8.1. As of 15 April 2014, IE 6 is no longer supported.

The driver supports running 32-bit and 64-bit versions of the browser. The choice of how to determine which "bit-ness" to use in launching the browser depends on which version of the IEDriverServer.exe is launched. If the 32-bit version of IEDriverServer.exe is launched, the 32-bit version of IE will be launched. Similarly, if the 64-bit version of IEDriverServer.exe is launched, the 64-bit version of IE will be launched.

Usage

WebDriver driver = new InternetExplorerDriver();

Pros

- Runs in a real browser and supports Javascript

Cons

- Obviously the InternetExplorerDriver will only work on Windows!
- Comparatively slow (though still pretty snappy!)

- **ChromeDriver**

ChromeDriver is maintained / supported by the Chromium project iteslf. WebDriver works with Chrome through the chromedriver binary (found on the chromium project's download page). You need to have both chromedriver and a version of chrome browser installed. chromedriver needs to be placed somewhere on your system's path in order for WebDriver to automatically discover it. The Chrome browser itself is discovered by chromedriver in the default installation path. These both can be overridden by environment variables. Please refer to the wiki for more information.

Usage

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

Pros

- Runs in a real browser and supports JavaScript
- Because Chrome is a Webkit-based browser, the ChromeDriver may allow you to verify that your site works in Safari. Note that since Chrome uses its own V8 JavaScript engine rather than Safari's Nitro engine, JavaScript execution may differ.

Cons

- Slower than the HtmlUnit Driver

Running Standalone Selenium Server for use with RemoteDrivers

From Selenium's Download page download selenium-server-standalone-<version>.jar and optionally IEDriverServer. If you plan to work with Chrome, download it from Google Code.

Unpack IEDriverServer and/or chromedriver and put them in a directory which is on the $PATH / %PATH% - the Selenium Server should then be able to handle requests for IE / Chrome without additional modifications.

Start the server on the command line with

```
java -jar <path_to>/selenium-server-standalone-<version>.jar
```

If you want to use native events functionality, indicate this on the command line with the option

```
-Dwebdriver.enable.native.events=1
```

For other command line options, execute

```
java -jar <path_to>/selenium-server-standalone-<version>.jar -help
```

WebDriver: Advanced Usage

Explicit and Implicit Waits

Waiting is having the automated task execution elapse a certain amount of time before continuing with the next step. You should choose to use Explicit Waits or Implicit Waits.

WARNING: Do not mix implicit and explicit waits. Doing so can cause unpredictable wait times. For example setting an implicit wait of 10 seconds and an explicit wait of 15 seconds, could cause a timeout to occur after 20 seconds.

Explicit Waits:

An explicit wait is code you define to wait for a certain condition to occur before proceeding further in the code. The worst case of this is Thread.sleep(), which sets the condition to an exact time period to wait. There are some convenience methods provided that help you write code that will wait only as long as required. WebDriverWait in combination with ExpectedCondition is one way this can be accomplished.

```
WebDriver driver = new FirefoxDriver();
driver.get("http://somedomain/url_that_delays_loading");
WebElementmyDynamicElement = (new WebDriverWait(driver, 10))
  .until(ExpectedConditions.presenceOfElementLocated(By.id("myDynamicElement")));
```

This waits up to 10 seconds before throwing a TimeoutException or if it finds the element will return it in 0 - 10 seconds. WebDriverWait by default calls the ExpectedCondition every 500 milliseconds until it returns successfully. A successful return value for the ExpectedCondition function type is a Boolean value of true, or a non-null object.

Expected Conditions

There are some common conditions that are frequently encountered when automating web browsers. Listed below are a few examples for the usage of such conditions. The Java, C#, and Python bindings include convienence methods so you don't have to code an ExpectedCondition class yourself or create your own utility package for them.

Element is Clickable - it is Displayed and Enabled.

```
WebDriverWait wait = new WebDriverWait(driver, 10);
WebElement element = wait.until(ExpectedConditions.elementToBeClickable(By.id("someid")));
```

The ExpectedConditions package (https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/support/ui/ExpectedConditions.html) contains a set of predefined conditions to use with WebDriverWait.

Implicit Waits

An implicit wait is to tell WebDriver to poll the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available. The default setting is 0. Once set, the implicit wait is set for the life of the WebDriver object instance.

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://somedomain/url_that_delays_loading");
WebElementmyDynamicElement = driver.findElement(By.id("myDynamicElement"));
```

RemoteWebDriver

An example for the same

```
importjava.io.File;
import java.net.URL;
importorg.openqa.selenium.OutputType;
importorg.openqa.selenium.TakesScreenshot;
importorg.openqa.selenium.WebDriver;
importorg.openqa.selenium.remote.Augmenter;
importorg.openqa.selenium.remote.DesiredCapabilities;
importorg.openqa.selenium.remote.RemoteWebDriver;

public class Testing {

public void myTest() throws Exception {
    WebDriver driver = new RemoteWebDriver(
new URL("http://localhost:4444/wd/hub"),
DesiredCapabilities.firefox());

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

    // RemoteWebDriver does not implement the TakesScreenshot class
    // if the driver does have the Capabilities to take a screenshot
    // then Augmenter will add the TakesScreenshot methods to the instance
    WebDriver augmentedDriver = new Augmenter().augment(driver);
    File screenshot = ((TakesScreenshot)augmentedDriver).
getScreenshotAs(OutputType.FILE);
  }
}
```

Using a FirefoxProfile

```
FirefoxProfilefp = new FirefoxProfile();
// set something on the profile...
DesiredCapabilities dc = DesiredCapabilities.firefox();
dc.setCapability(FirefoxDriver.PROFILE, fp);
WebDriver driver = new RemoteWebDriver(dc);
```

Using ChromeOptions

```
ChromeOptions options = new ChromeOptions();
// set some options
DesiredCapabilities dc = DesiredCapabilities.chrome();
dc.setCapability(ChromeOptions.CAPABILITY, options);
WebDriver driver = new RemoteWebDriver(dc);
```

JavaScript and Selenium JavaScriptExecutor

What is JavaScript?

JavaScript is the preferred language inside the browser to interact with HTML dom. This means that a Browser has JavaScript implementation in it and understands the JavaScript commands. You can disable it using browser options in your browser.JavaScript is widely used for client-side web development

JavaScript was also the language that was used by earlier Selenium versions, it is still used by Selenium webdriver to perform some actions. For e.g. Selenium has Xpath implemented in JavaScript for IE, to overcome the lack of Xpath engine in IE. Understanding JavaScript is fun and it can enable you to do lot of cool things that otherwise you may find tricky.

WebDriver gives you a method called Driver.executeScript which executes the JavaScript in context of the loaded browser page.

First thing to know that the JavaScriptExecutor comes separately and also comes under the WebDriver but both do the same thing. Within the WebDriver, it is named as ExecuteScript. Below you will find the examples for both. You can use any of it. Well practically anything you want to do in browser.

The tag we insert an JavaScript in HTML page is <script type="text/javascript"></script>

Evaluating Xpaths

You can use javascripts to find an element by Xpath. This is a brilliant way to skip internal Xpath engines. Here is what you have to do

```
WebElement value = (WebElement) Driver.executeScript("return document.evaluate(
'//body//div/iframe' ,document, null, XPathResult.FIRST_ORDERED_NODE_TYPE,
null).singleNodeValue;");
```

Notice the command document.evaluate(). This is the Xpath evaluator in javascript. Signature of the function is

document.evaluate(xpathExpression, contextNode, namespaceResolver, resultType, result);

xpathExpression: A string containing the XPath expression to be evaluated.

contextNode: A node in the document against which the xpathExpression should be evaluated, including any and all of its child nodes. The document node is the most commonly used.

NamespaceResolver: A function that will be passed any namespace prefixes contained within xpathExpression which returns a string representing the namespace URI associated with that prefix.

resultType: A constant that specifies the desired result type to be returned as a result of the evaluation. The most commonly passed constant is XPathResult.ANY_TYPE which will return the results of the XPath expression as the most natural type.

result: If an existing XPathResult object is specified, it will be reused to return the results. Specifying null will create a new XPathResult object

Finding Element

Using selenium we can find any element on page using something similar to

Driver.findElementById("Id of the element");

we can do the same thing using JavaScript like so

WebElementsearchbox = null;

searchbox = (WebElement) ff.executeScript("return document.getElementById('gsc-i-id1');", searchbox);

you will get the element in the searchbox variable.

Finding elements Selenium command

List\<WebElement\> elements = List\<WebElement\> Driver.findElementByTagName("div");

using JavaScript

List elements = (List') ff.executeScript("return document.getElementsByTagName('div');", searchbox);

Changing style attribute of the elements

You can change the style property of elements to change the rendered view of the element.creating border

Here is the code.

Driver.executeScript("document.getElementById('text-4').style.borderColor = 'Red'");

Coloring elements can also help us take screenshots with visual markers to identify problematic elements.

Getting Element Attributes

You can get any value of valid attributes of the element. For eg

```
Driver.findElementById("gsc-i-id1").getAttribute("Class");
```

this code will get the value of class attribute of an element with id = gsc-i-id1.

In Javascript same thing can be executed using

```
String className = Driver.executeScript("return document.getElementById('gsc-i-id1').getAttribute('class');"));
```

Size of Window

Finding the size of inner browser window. It is the size of the window in which you see the web page.

```
Driver.executeScript("return window.innerHeight;")
```

What is JavaScriptExecutor?

JavaScriptExecutor is an interface which provides mechanism to execute Javascript through selenium driver. It provides “executescript” & “executeAsyncScript” methods, to run JavaScript in the context of the currently selected frame or window.

Generate Alert Pop Window

```
JavascriptExecutorjs = (JavascriptExecutor)driver;

Js.executeScript("alert('hello world');");
```

Click Action

```
JavascriptExecutorjs = (JavascriptExecutor)driver;
js.executeScript("arguments[0].click();", element);
```

Refresh Browser

```
JavascriptExecutorjs = (JavascriptExecutor)driver;
driver.executeScript("history.go(0)");
```

Get InnerText of a Webpage

```
JavascriptExecutorjs = (JavascriptExecutor)driver;
stringsText =  js.executeScript("return document.documentElement.innerText;").toString();
```

Get Title of a WebPage

```
JavascriptExecutorjs = (JavascriptExecutor)driver;
stringsText =  js.executeScript("return document.title;").toString();
```

Scroll Page

```
JavascriptExecutorjs = (JavascriptExecutor)driver;
 //Vertical scroll - down by 150  pixels
js.executeScript("window.scrollBy(0,150)");
```

Similarly you can execute practically any JavaScript command using Selenium. I hope this blog gives you an idea of what can be done and you can take it forward by playing around with different JavaScripts.

Selenium Grid

What is Selenium-Grid?

Selenium-Grid allows you run your tests on different machines against different browsers in parallel. That is, running multiple tests at the same time against different machines running different browsers and operating systems. Essentially, Selenium-Grid support distributed test execution. It allows for running your tests in a distributed test execution environment.

When to Use It

Generally speaking, there's two reasons why you might want to use Selenium-Grid.

To run your tests against multiple browsers, multiple versions of browser, and browsers running on different operating systems.

To reduce the time it takes for the test suite to complete a test pass.

Selenium-Grid is used to speed up the execution of a test pass by using multiple machines to run tests in parallel. For example, if you have a suite of 100 tests, but you set up Selenium-Grid to support 4 different machines (VMs or separate physical machines) to run those tests, your test suite will complete in (roughly) one-fourth the time as it would if you ran your tests sequentially on a single machine. For large test suites, and long-running test suite such as those performing large amounts of data-validation, this can be a significant time-saver. Some test suites can take hours to run. Another reason to boost the time spent running the suite is to shorten the turnaround time for test results after developers check-in code for the AUT. Increasingly software teams practicing Agile software development want test feedback as immediately as possible as opposed to wait overnight for an overnight test pass.

Selenium-Grid is also used to support running tests against multiple runtime environments, specifically, against different browsers at the same time. For example, a 'grid' of virtual machines can be setup with each supporting a different browser that the application to be tested must support. So, machine 1 has Internet Explorer 8, machine 2, Internet Explorer 9, machine 3 the latest Chrome, and machine 4 the latest Firefox. When the test suite is run, Selenium-Grid receives each test-browser combination and assigns each test to run against its required browser.

In addition, one can have a grid of all the same browser, type and version. For instance, one could have a grid of 4 machines each running 3 instances of Firefox 12, allowing for a 'server-farm' (in a sense) of available Firefox instances. When the suite runs, each test is passed to Selenium-Grid which assigns the test to the next available Firefox instance. In this manner one gets test pass where conceivably 12 tests are all running at the same time in parallel, significantly reducing the time required to complete a test pass.

Selenium-Grid is very flexible. These two examples can be combined to allow multiple instances of each browser type and version. A configuration such as this would provide both, parallel execution for fast test pass completion and support for multiple browser types and versions simultaneously.

Selenium-Grid 2.0

Selenium-Grid 2.0 is the latest release as of the writing of this document (5/26/2012). It is quite different from version 1 of Selenium-Grid. In 2.0 Selenium-Grid was merged with the Selenium-RC server. Now, you only need to download a single .jar file to get the remote Selenium-RC-Server and Selenium-Grid all in one package.

How Selenium-Grid Works–With a Hub and Nodes

A grid consists of a single hub, and one or more nodes. Both are started using the selenium-server.jar executable. We've listed some examples in the following sections of this chapter.

The hub receives a test to be executed along with information on which browser and 'platform' (i.e. WINDOWS, LINUX, etc) where the test should be run. It 'knows' the configuration of each node that has been 'registered' to the hub. Using this information it selects an available node that has the requested browser-platform combination. Once a node has been selected, Selenium commands initiated by the test are sent to the hub, which passes them to the node assigned to that test. The node runs the browser, and executes the Selenium commands within that browser against the application under test.

Starting Selenium-Grid

Generally you would start a hub first since nodes depend on a hub. This is not absolutely necessary however, since nodes can recognize when a hub has been started and vice-versa. For learning purposes though, it would easier to start the hub first, otherwise you'll see error messages that may not want to start off with your first time using Selenium-Grid.

Starting a Hub

To start a hub with default parameters, run the following command from a command-line shell. This will work on all the supported platforms, Windows Linux, or Mac OSX.

```
java -jar selenium-server-standalone-2.44.0.jar -role hub
```

This starts a hub using default parameter values. We'll explain these parameters in following subsections. Note that you will likely have to change the version number in the jar filename depending on which version of the selenium-server you're using.

Starting a Node

To start a node using default parameters, run the following command from a command-line.

```
java -jar selenium-server-standalone-2.44.0.jar -role node  -hub http://localhost:4444/grid/register
```

This assumes the hub has been started above using default parameters. The default port the hub uses to listen for new requests is port 4444. This is why port 4444 was used in the URL for locating the hub. Also the use of 'localhost' assumes your node is running on the same machine as your hub. For getting started this is probably easiest. If running the hub and node on separate machines, simply replace 'localhost' with the hostname of the remote machine running the hub.

Hub and Node Configuration

To run the hub using the default options simply specify -role hub to the Selenium-Server

```
java -jar selenium-server-standalone-2.44.0.jar -hub
```

You should see the following logging output.

Jul 19, 2012 10:46:21 AM org.openqa.grid.selenium.GridLauncher main
INFO: Launching a selenium grid server
2012-07-19 10:46:25.082:INFO:osjs.Server:jetty-7.x.y-SNAPSHOT
2012-07-19 10:46:25.151:INFO:osjsh.ContextHandler:startedo.s.j.s.ServletContextHandler{/,null}
2012-07-19 10:46:25.185:INFO:osjs.AbstractConnector:Started SocketConnector@0.0.0.0:4444

Specifying the Port
The default port used by the hub is 4444. The port being referred to here, is the TCP/IP port used when the 'client', that is, the automated tests connect to the Selenium-Grid hub. If another application on your computer is already using this port, or if, you already have a Selenium-Server started, you'll see the following message in the log output.

10:56:35.490 WARN - Failed to start: SocketListener0@0.0.0.0:4444
Exception in thread "main" java.net.BindException: Selenium is already running on port 4444. Or some other service is.

If this occurs you can either shutdown the other process that is using port 4444, or you can tell Selenium-Grid to use a different port for its hub. Use the -port option for changing the port used by the hub.

```
java -jar selenium-server-standalone-2.44.0.jar -role hub -port 4441
```

This will work even if another hub is already running on the same machine, that is, as long as they're both not using port 4441.
You may, however, want to see what process is using port 4444 so you can allow the hub to use the default. To see the ports used by all running programs on your machine use the command.

```
netstat -a
```

This should work on all supported systems, Unix/Linux, MacOs, and Windows although additional options beyond -a may be required. Basically you need to display the process ID along with the port. In Unix you may 'grep' the output (use a pipe) from the port number to only display those records you're concerned with.

TestNG

Introduction

TestNG is a testing framework designed to simplify a broad range of testing needs, from unit testing (testing a class in isolation of the others) to integration testing (testing entire systems made of several classes, several packages and even several external frameworks, such as application servers).
Writing a test is typically a three-step process:

- Write the business logic of your test and insert TestNG annotations in your code.
- Add the information about your test (e.g. the class name, the groups you wish to run, etc...) in a testng.xml file or in build.xml.
- Run TestNG.

The concepts are as follows:

- A suite is represented by one XML file. It can contain one or more tests and is defined by the <suite> tag.
- A test is represented by <test> and can contain one or more TestNG classes.
- A TestNG class is a Java class that contains at least one TestNG annotation. It is represented by the <class> tag and can contain one or more test methods.
- A test method is a Java method annotated by @Test in your source.

A TestNG test can be configured by @BeforeXXX and @AfterXXX annotations which allows to perform some Java logic before and after a certain point, these points being either of the items listed above.

Annotations

Here is a quick overview of the annotations available in TestNG along with their attributes.

@BeforeSuite @AfterSuite @BeforeTest @AfterTest @BeforeGroups @AfterGroups @BeforeClass @AfterClass @BeforeMethod @AfterMethod	Configuration information for a TestNG class: @BeforeSuite: The annotated method will be run before all tests in this suite have run. @AfterSuite: The annotated method will be run after all tests in this suite have run. @BeforeTest: The annotated method will be run before any test method belonging to the classes inside the <test> tag is run. @AfterTest: The annotated method will be run after all the test methods belonging to the classes inside the <test> tag have run. @BeforeGroups: The list of groups that this configuration method will run before. This method is guaranteed to run shortly before the first test method that belongs to any of these groups is invoked. @AfterGroups: The list of groups that this configuration method will run after. This method is guaranteed to run shortly after the last test method that belongs to any of these groups is invoked. @BeforeClass: The annotated method will be run before the first test method in the current class is invoked. @AfterClass: The annotated method will be run after all the test methods in the current class have been run. @BeforeMethod: The annotated method will be run before each test method. @AfterMethod: The annotated method will be run after each test method.
alwaysRun	For before methods (beforeSuite, beforeTest, beforeTestClass and beforeTestMethod, but not beforeGroups): If set to true, this configuration method will be run regardless of what groups it belongs to. For after methods (afterSuite, afterClass, ...): If set to true, this configuration method will be run even if one or more methods invoked previously failed or was skipped.

dependsOnGroups	The list of groups this method depends on.
dependsOnMethods	The list of methods this method depends on.
enabled	Whether methods on this class/method are enabled.
groups	The list of groups this class/method belongs to.
inheritGroups	If true, this method will belong to groups specified in the @Test annotation at the class level.
@DataProvider	Marks a method as supplying data for a test method. The annotated method must return an Object[][] where each Object[] can be assigned the parameter list of the test method. The @Test method that wants to receive data from this DataProvider needs to use a dataProvider name equals to the name of this annotation.
name	The name of this data provider. If it's not supplied, the name of this data provider will automatically be set to the name of the method.
parallel	If set to true, tests generated using this data provider are run in parallel. Default value is false.
@Factory	Marks a method as a factory that returns objects that will be used by TestNG as Test classes. The method must return Object[].
@Listeners	Defines listeners on a test class.
value	An array of classes that extend org.testng.ITestNGListener.
@Parameters	Describes how to pass parameters to a @Test method.
value	The list of variables used to fill the parameters of this method.
@Test	Marks a class or a method as part of the test.
alwaysRun	If set to true, this test method will always be run even if it depends on a method that failed.
dataProvider	The name of the data provider for this test method.
dataProviderClass	The class where to look for the data provider. If not specified, the data provider will be looked on the class of the current test method or one of its base classes. If this attribute is specified, the data provider method needs to be static on the specified class.
dependsOnGroups	The list of groups this method depends on.
dependsOnMethods	The list of methods this method depends on.
description	The description for this method.

	enabled	Whether methods on this class/method are enabled.
	expectedExceptions	The list of exceptions that a test method is expected to throw. If no exception or a different than one on this list is thrown, this test will be marked a failure.
	groups	The list of groups this class/method belongs to.
	invocationCount	The number of times this method should be invoked.
	invocationTimeOut	The maximum number of milliseconds this test should take for the cumulated time of all the invocationcounts. This attribute will be ignored if invocationCount is not specified.
	priority	The priority for this test method. Lower priorities will be scheduled first.
	successPercentage	The percentage of success expected from this method
	singleThreaded	If set to true, all the methods on this test class are guaranteed to run in the same thread, even if the tests are currently being run with parallel="methods". This attribute can only be used at the class level and it will be ignored if used at the method level. Note: this attribute used to be called sequential (now deprecated).
	timeOut	The maximum number of milliseconds this test should take.
	threadPoolSize	The size of the thread pool for this method. The method will be invoked from multiple threads as specified by invocationCount. Note: this attribute is ignored if invocationCount is not specified

testng.xml

You can invoke TestNG in several different ways:

- With a testng.xml file
- With ant
- From the command line

Here is an example testng.xml file:

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Suite1" verbose="1" >
<test name="Nopackage" >
<classes>
<class name="NoPackageTest" />
</classes>
</test>
<test name="Regression1">
<classes>
<class name="test.sample.ParameterSample"/>
<class name="test.sample.ParameterTest"/>
</classes>
</test>
</suit>
```

You can specify package names instead of class names:

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Suite1" verbose="1" >
<test name="Regression1"   >
<packages>
<package name="test.sample" />
</packages>
</test>
</suite>
```

In this example, TestNG will look at all the classes in the package test.sample and will retain only classes that have TestNG annotations.
You can also specify groups and methods to be included and excluded:

```
<test name="Regression1">
<groups>
<run>
<exclude name="brokenTests"  />
<include name="checkinTests"  />
</run>
</groups>
<classes>
<class name="test.IndividualMethodsTest">
<methods>
<include name="testMethod" />
</methods>
</class>
</classes>
</test>
```

You can also define new groups inside testng.xml and specify additional details in attributes, such as whether to run the tests in parallel, how many threads to use, whether you are running JUnit tests, etc...
By default, TestNG will run your tests in the order they are found in the XML file. If you want the classes and methods listed in this file to be run in an unpredictable order, set the preserve-order attribute to false

```
<test name="Regression1" preserve-order="false">
<classes>
<class name="test.Test1">
<methods>
<include name="m1" />
<include name="m2" />
</methods>
</class>
<class name="test.Test2" />
</classes>
</test>
```

Running TestNG

TestNG can be invoked in different ways:

- Command line
  Assuming that you have TestNG in your class path, the simplest way to invoke TestNG is as follows:
  You need to specify at least one XML file describing the TestNG suite you are trying to run.

java org.testng.TestNG testng1.xml [testng2.xml testng3.xml ...]

- ant
  You define the TestNG ant task as follows. Refer http://testng.org/doc/ant.html for more information.

  <taskdef resource="testngtasks" classpath="testng.jar"/>

- Eclipse: refer http://testng.org/doc/eclipse.html for more information

Test methods, Test classes and Test groups

Test methods:
Test methods are annotated with @Test. Methods annotated with @Test that happen to return a value will be ignored, unless you set allow-return-values to true in your testng.xml:
<suite allow-return-values="true">
or
<test allow-return-values="true">

Test groups:

TestNG allows you to perform sophisticated groupings of test methods. Not only can you declare that methods belong to groups, but you can also specify groups that contain other groups. Then TestNG can be invoked and asked to include a certain set of groups (or regular expressions) while excluding another set. This gives you maximum flexibility in how you partition your tests and doesn't require you to recompile anything if you want to run two different sets of tests back to back.
Groups are specified in your testng.xml file and can be found either under the <test> or <suite> tag. Groups specified in the <suite> tag apply to all the <test> tags underneath. Note that groups are accumulative in these tags: if you specify group "a" in <suite> and "b" in <test>, then both "a" and "b" will be included.
For example, it is quite common to have at least two categories of tests

- Check-in tests. These tests should be run before you submit new code. They should typically be fast and just make sure no basic functionality was broken.
- Functional tests. These tests should cover all the functionalities of your software and be run at least once a day, although ideally you would want to run them continuously.

Typically, check-in tests are a subset of functional tests. TestNG allows you to specify this in a very intuitive way with test groups. For example, you could structure your test by saying that your entire test class belongs to the "functest" group, and additionally that a couple of methods belong to the group "checkintest":

```
public class Test1 {
@Test(groups = { "functest", "checkintest" })
public void testMethod1() {
  }
@Test(groups = {"functest", "checkintest"} )
public void testMethod2() {
  }
@Test(groups = { "functest" })
public void testMethod3() {
  }
}
```

Invoking TestNG with

```
<test name="Test1">
<groups>
<run>
<include name="functest"/>
</run>
</groups>
<classes>
<class name="example1.Test1"/>
</classes>
</test>
```

will run all the test methods in that classes, while invoking it with checkintest will only run testMethod1() and testMethod2().

Groups of groups:

Groups can also include other groups. These groups are called "MetaGroups". For example, you might want to define a group "all" that includes "checkintest" and "functest". "functest" itself will contain the groups "windows" and "linux" while "checkintest will only contain "windows". Here is how you would define this in your property file:

```
<test name="Regression1">
<groups>
<define name="functest">
<include name="windows"/>
<include name="linux"/>
</define>

<define name="all">
<include name="functest"/>
<include name="checkintest"/>
</define>
```

```
<run>
<include name="all"/>
</run>
</groups>
<classes>
<class name="test.sample.Test1"/>
</classes>
</test>
```

Parameters:

Test methods don't have to be parameterless. You can use an arbitrary number of parameters on each of your test method, and you instruct TestNG to pass you the correct parameters with the @Parameters annotation.
There are two ways to set these parameters: with testng.xml or programmatically.
Parameters from testng.xml
If you are using simple values for your parameters, you can specify them in your testng.xml:

```
@Parameters({ "first-name" })
@Test
public void testSingleString(String firstName) {
System.out.println("Invoked testString " + firstName);

assert "Cedric".equals(firstName);
}
```

In this code, we specify that the parameter firstName of your Java method should receive the value of the XML parameter called first-name. This XML parameter is defined in testng.xml:

```
<suite name="My suite">
<parameter name="first-name"  value="Cedric"/>

<test name="Simple example">
<-- ... -->
```

The same technique can be used for @Before/After and @Factory annotations:

```
@Parameters({ "datasource", "jdbcDriver" })
@BeforeMethod

public void beforeTest(String ds, String driver) {
m_dataSource = ...;                    // look up the value of datasource

m_jdbcDriver = driver;
}
```

Parameters with DataProviders:

Specifying parameters in testng.xml might not be sufficient if you need to pass complex parameters, or parameters that need to be created from Java (complex objects, objects read from a property file or a database, etc...). In this case, you can use a Data Provider to supply the values you need to test. A Data Provider is a method on your class that returns an array of array of objects.

This method is annotated with @DataProvider:

```
//This method will provide data to any test method that declares that its Data Provider
//is named "test1"
@DataProvider(name = "test1")
public Object[][] createData1() {
return new Object[][] {
{ "Cedric", new Integer(36) },
{ "Anne", new Integer(37)},
 };
}
//This test method declares that its data should be supplied by the Data Provider
//named "test1"

@Test(dataProvider = "test1")
public void verifyData1(String n1, Integer n2) {
System.out.println(n1 + " " + n2);
}
```

Output is
Cedric 36
Anne 37

A @Test method specifies its Data Provider with the dataProvider attribute. This name must correspond to a method on the same class annotated with @DataProvider(name="...") with a matching name.
By default, the data provider will be looked for in the current test class or one of its base classes. If you want to put your data provider in a different class, it needs to be a static method or a class with a non-arg constructor, and you specify the class where it can be found in the dataProviderClass attribute:

```
public class StaticProvider {
@DataProvider(name = "create")
public static Object[][] createData() {
return new Object[][] {
new Object[] { new Integer(42) }
   };
  }
}

public class MyTest {
```

```
@Test(dataProvider = "create", dataProviderClass = StaticProvider.class)
public void test(Integer n) {
   // ...
  }
}
```

Parallelism and time-outs:

You can instruct TestNG to run your tests in separate threads in various ways.

Parallel suites:

This is useful if you are running several suite files (e.g. "java org.testng.TestNG testng1.xml testng2.xml") and you want each of these suites to be run in a separate thread.
You can use the following command line flag to specify the size of a thread pool:

```
javaorg.testng.TestNG -suitethreadpoolsize 3 testng1.xml testng2.xml testng3.xml
```

The corresponding ant task name is suitethreadpoolsize.

Parallel tests, classes and methods:

The parallel attribute on the <suite> tag can take one of following values:

```
<suite name="My suite" parallel="methods" thread-count="5">
<suite name="My suite" parallel="tests" thread-count="5">
<suite name="My suite" parallel="classes" thread-count="5">
<suite name="My suite" parallel="instances" thread-count="5">
```

parallel="methods": TestNG will run all your test methods in separate threads. Dependent methods will also run in separate threads but they will respect the order that you specified.

parallel="tests": TestNG will run all the methods in the same <test> tag in the same thread, but each <test> tag will be in a separate thread. This allows you to group all your classes that are not thread safe in the same <test> and guarantee they will all run in the same thread while taking advantage of TestNG using as many threads as possible to run your tests.

parallel="classes": TestNG will run all the methods in the same class in the same thread, but each class will be run in a separate thread.

parallel="instances": TestNG will run all the methods in the same instance in the same thread, but two methods on two different instances will be running in different threads.

Additionally, the attribute thread-count allows you to specify how many threads should be allocated for this execution.

Note: the @Test attribute timeOut works in both parallel and non-parallel mode.

You can also specify that a @Test method should be invoked from different threads. You can use the attribute threadPoolSize to achieve this result:

```
@Test(threadPoolSize = 3, invocationCount = 10,  timeOut = 10000)
public void testServer() {
}
```

In this example, the function testServer will be invoked ten times from three different threads. Additionally, a time-out of ten seconds guarantees that none of the threads will block on this thread forever.

Apache Ant?

While creating a complete software product, one needs to take care different third party API, their classpath, cleaning previous executable binary files, compiling our source code, execution of source code, creation of reports and deployment code base etc. If these tasks are done one by one manually, it will take an enormous time, and the process will be prone to errors.

Here comes the importance of a build tool like Ant. It stores, executes and automates all process in a sequential order mentioned in Ant's configuration file (usually build.xml).

Benefit of Ant build

- Ant creates the application life cycle i.e. clean, compile, set dependency, execute, report, etc.
- Third party API dependency can be set by Ant i.e. other Jar file's class path is set by Ant build file.
- A complete application is created for End to End delivery and deployment.
- It is a simple build tool where all configurations can be done using XML file and which can be executed from the command line.
- It makes your code clean as configuration is separate from actual application logic.

Understanding Build.xml

Build.xml is the most important component of Ant build tool.

For a Java project, all cleaning, setup, compilation and deployment related task are mentioned in this file in XML format. When we execute this XML file using command line or any IDE plugin, all instructions written into this file will get executed in sequential manner.

Let's understand the code within a sample build.XML

- Project tag is used to mention a project name and basedir attribute.
    - The basedir is the root directory of an application and is one of properties of the Ant.
    - Similarly ant.java.version, ant.version and ant.home are properties of Ant.

<project name="YTMonetize" basedir=".">

- Property tags are used as variables in build.XML file to be used in further steps

  <property name="build.dir" value="${basedir}/build"/>

  <property name="external.jars" value=".\resources"/>

  <property name="ytoperation.dir" value="${external.jars}/YTOperation"/>

  <property name="src.dir"value="${basedir}/src"/>

- Target tags used as steps that will execute in sequential order. Name attribute is the name of the target. You can have multiple targets in a single build.xml

  <target name="setClassPath">

- path tag is used to bundle all files logically which are in the common location

  <path id="classpath_jars">

- pathelement tag will set the path to the root of common location where all files are stored

  <pathelement path="${basedir}/"/>

- pathconvert tag used to convert paths of all common file inside path tag to system's classpath format

  <pathconvertpathsep=";" property="test.classpath" refid="classpath_jars"/>

- fileset tag used to set classpath for different third party jar in our project

  <filesetdir="${ytoperation.dir}" includes="*.jar"/>

- Echo tag is used to print text on the console

  <echo message="deleting existing build directory"/>

- Delete tag will clean data from given folder

  <delete dir="${build.dir}"/>

- mkdir tag will create a new directory

  <mkdirdir="${build.dir}"/>

- javac tag used to compile java source code and move .class files to a new folder

<javacdestdir="${build.dir}" srcdir="${src.dir}">

  <classpathrefid="classpath_jars"/></javac>

- jar tag will create jar file from .class files

<jar destfile="${ytoperation.dir}/YTOperation.jar" basedir="${build.dir}">

- manifest tag will set your main class for execution

  <manifest>

  <attribute name="Main-Class" value="test.Main"/>

  </manifest>

- 'depends' attribute used to make one target to depend on another target

  <target name="run" depends="compile">

- java tag will execute main function from the jar created in compile target section

  <java jar="${ytoperation.dir}/YTOperation.jar" fork="true"/>

Maven & Jenkins with Selenium

What is Jenkins?

Jenkins is the leading open-source continuous integration tool developed by Hudson lab. It is cross-platform and can be used on Windows, Linux, Mac OS and Solaris environments. Jenkins is written in Java. Jenkin's chief usage is to monitor any job which can be SVN checkout, cron or any application states. It fires pre-configured actions when a particular step occurs in jobs.

Important Features of Jenkins

Change Support: Jenkins generates the list of all changes done in repositories like SVN.
Permanent links: Jenkins provides direct links to the latest build or failed build that can be used for easy communication
Installation: Jenkins is easy to install either using direct installation file (exe) or war file to deploy using application server.
Email integration: Jenkins can be configured to email the content of the status of the build.
Easy Configuration: To configure various tasks on Jenkins is easy.
TestNG test: Jenkins can be configured to run the automation test build on TestNG after each build of SVN.
Multiple VMs: Jenkins can be configured to distribute the build on multiple machines.
Project build: Jenkins documents the details of jar, version of jar and mapping of build and jar numbers.
Plugins: 3rd party plugin can be configured in Jenkins to use features and additional functionality.

Why Jenkins and Selenium?

Running Selenium tests in Jenkins allows you to run your tests every time your software changes and deploy the software to a new environment when the tests pass.

Jenkins can schedule your tests to run at specific time.
You can save the execution history and Test Reports.
Jenkins supports Maven for building and testing a project in continuous integration.

Why Maven & Jenkins

Selenium WebDriver is great for browser automation. But, when using it for testing and building a test framework, it feels underpowered. Integrating Maven with Selenium provides following benefits, Apache Maven provides support for managing the full lifecycle of a test project.

- Maven is used to define project structure, dependencies, build, and test management.
- Using pom.xml(Maven) you can configure dependencies needed for building testing and running code.
- Maven automatically downloads the necessary files from the repository while building the project.

Refer links for more information.
https://www.seleniumeasy.com/maven-tutorials
and
http://www.guru99.com/maven-jenkins-with-selenium-complete-tutorial.html

Java

For Java concepts refer http://www.headfirstlabs.com/books/hfjava/

Introduction to Collections

A collection — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.

What Is a Collections Framework?

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- Interfaces: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

- Implementations: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

- Algorithms: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve.

Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

Reduces programming effort: By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.

Increases program speed and quality: This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.

Allows interoperability among unrelated APIs: The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.

Reduces effort to learn and to use new APIs: Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.

Reduces effort to design new APIs: This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.

Fosters software reuse: New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Interfaces

The core collection interfaces encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework.

Note that all the core collection interfaces are generic. For example, this is the declaration of the Collection interface.

public interface Collection<E>...

The <E> syntax tells you that the interface is generic. When you declare a Collection instance you can and should specify the type of object contained in the collection. Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime.

The following list describes the core collection interfaces:

- Collection — the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

- Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine..

- List — an ordered collection (sometimes called a sequence). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor of List.

- Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering? Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties. Also see The Queue Interface section.

- Deque — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations. Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends.

- Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map.

The last two core collection interfaces are merely sorted versions of Set and Map:

- SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

- SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements.It extends AbstractList class and implements List interface.

Java ArrayList class can contain duplicate elements.
Java ArrayList class maintains insertion order.
Java ArrayList class is non synchronized.
Java ArrayList allows random access because array works at the index basis.
In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.
Java Non-generic Vs Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

```
ArrayList al=new ArrayList();       //creating old non-generic arraylist
```

Let's see the new generic example of creating java collection.

```
ArrayList<String> al=new ArrayList<String>();      //creating new generic arraylist
```

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives compile time error.

Example of Java ArrayList class

```
importjava.util.*;
class TestCollection1{
public static void main(String args[]){

ArrayList<String> al=new ArrayList<String>();//creating arraylist
al.add("Ravi");//adding object in arraylist
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");

  Iterator itr=al.iterator();//getting Iterator from arraylist to traverse elements
while(itr.hasNext()){
System.out.println(itr.next());
  }
 }
}
```

Output is

```
Ravi
Vijay
Ravi
Ajay
```

Two ways to iterate the elements of collection in java

- By Iterator interface.
- By for-each loop.

In the above example, we have seen traversing ArrayList by Iterator. Let's see the example to traverse ArrayList elements using for-each loop.

```
importjava.util.*;
class TestCollection2{
public static void main(String args[]){
ArrayList<String> al=new ArrayList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
for(String obj:al)
System.out.println(obj);
 }
}
```

Below table summarize the discussion on collection framework.

	Collection Class name	Ordered	Sorted
Map	Hashtable	No	No
	HashMap	No	No
	TreeMap	Sorted	By natural order or custom order
	LinkedHashMap	By insertion order or last access order	No
Set	HashSet	No	No
	TreeSet	Sorted	By natural order or custom order
	LinkedHashSet	By insertion order	No
List	ArrayList	Indexed	No
	Vector	Indexed	No
	LinkedList	Indexed	No
	Priority queue	Sorted	By to-do order