Writing sample from Carrie Schwalbe (all rights reserved)

This is a section of my Selenium framework cheat-sheet designed for entry-level automation engineers. It assumes a moderate level of Java experience and knowledge of Selenium, but not enough Selenium experience to quickly recall common methods or techniques. It is a review sheet, not a tutorial.

(Please note that the public sample might be truncated.)

Thank you for your consideration.

paneljump@yahoo.com

510 725-5846

Table of Contents

Selenium Project Quick Notes

These instructions are intended for Java programmers using Eclipse and Selenium with Java Standalone Server (Server Binding) only.

Part 1: Selenium Basics

What is Selenium?

Selenium is an API (Application Programming Interface). In Java, it is a library or series of libraries which allow Java programs to drive web pages. Selenium can be used with many programming languages (Java, C#, Python, etc.) and with many different browsers.

The main Selenium interface used for testing is WebDriver. It represents an idealized web browser, and is used to control the web browser itself and select elements in web pages. Some methods can be used as debugging aids, but these are beyond the scope of this paper at this time.

Adding Selenium to a Java Project

1. Go to the Selenium website http://docs.seleniumhq.org/download and download the Selenium Standalone Server. Save the JAR file.
2. Right-click on the project and select Properties at the bottom of the menu. "Properties" can also be found under the Project menu.
3. Click on Java Build Path in the left menu, select the Libraries tab if necessary, and click "Add External JARs..."
4. Navigate to the saved JAR file, select, and click Open.

Notice that the Javadocs may be included in the Selenium standalone server. If they are not, or if you choose to use Java client binding, you can download the Javadocs from the same download page. They will be listed near the Java client binding download link.

Adding Browser-Specific Driver Capabilities

The Selenium Standalone Server has built-in support for the Firefox driver. If you wish to test with other popular browsers (such as Safari, IE, or Chrome), additional steps may be required. These will be discussed in the Selenium Layer section.

Creating a Driver

What is a driver?

A driver is an object that can open and close multiple windows within a given browser. It can simulate common user actions by locating and acting upon web elements. The Selenium libraries, and third-party libraries, contain many useful classes such as FirefoxDriver, ChromeDriver, InternetExplorerDriver, and more. All of these classes implement one common interface, WebDriver, which require them to have many useful methods, such as getCurrentURL(), getWindowHandle(), and findElement(), in common.

NOTE: In practice, most automated web tests must be performed on multiple browser types. Code should be re-used, rather than copied and edited, whenever possible, so it is strongly recommended that drivers be declared to be of type WebDriver, the common interface, rather than by the more specific browser class.

Instantiating a Driver

This simple standalone class illustrates how to create a driver object:

```
package selenium;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class simpleDriver {

	public static void main(String[] args){

		WebDriver driver = new FirefoxDriver();
	}

}
```

Changing the Driver Target

WebElements cannot be located unless the driver is pointing to the proper target. Drivers can control many windows, but they only point to one at a time. Additionally, drivers can only point to one frame at a time, when frames exist on the page, and they must be explicitly directed to active modal dialogs such as Alerts. Instructions for setting the driver target can be found in the corresponding section within The Web Page Layer.

Web Element Locating Strategies

The findElement method, which acts on a WebDriver or a WebElement, allows for many different locating strategies. Elements can be identified by their id, name, tagname, xpath, css selector, and more.

```
(General)    WebElement findElement(By by);
(Specific)   WebElement findElement(By.id("onlyButton"))
```

The abstract class By has several nested concrete classes. In the example above, an instance of the static concrete class ById (wrapped in the outer class By) is returned when the method id(“onlyButton”) is enacted on By.

Note that findElement returns the first match on the page. To retrieve all matches, use findElements:

```
List<WebElement> findElements(By by);
```

The easiest way to identify a web element in the HTML on a web page is to start FireBug (a plugin for Firefox), click the HTML tab, click on the Element Inspector, and then click on the element on the page. This highlights the corresponding HTML. The best locating strategy for any given tag depends on its attributes and whether the element is static or dynamic.

Location and selection strategies for simple web elements are:

1. By.id(String str) is probably the most common method for identifying web elements.
2. By.name(String str) is the next most preferred method. By.tagName(String str), and By.className(String str) are similar.
3. By.linkText(String str) and By.partialLinkText(String str) are helpful for identifying links because developers rarely give names/IDs to anchor tags.
4. By.xpath(String str) This is useful when there is no suitable name or ID attribute, and when the elements are members of a dynamic table.
5. By.cssSelector(String str)

Xpaths

Xpath is the language used for locating nodes in XML docs (HTML is in the XML family). There are several ways to determine an expression for an xpath:

- Absolute path is the full path from the beginning of the page. This is not recommended because it can stop working if the element is moved, or if other elements in the path are changed.
  - The syntax looks like this: "/html/body/div[i]/..." Notice that absolute xpaths begin with a single '/'.
- Relative path expressions begin with a parent tag that can be determined by id or name. These can be more robust than absolute paths, especially when the parent element and element of interest share a block of code that is unlikely to be disturbed.
  - The syntax for a relative path looks like this: "//tag[@id='email']/div[i]/..." Notice that relative xpaths begin with two '/'.
  - This is how table cells, which rarely contain unique attributes, are typically found. For example:
    - "//table[@id='puzzle']/tbody/tr[3]/td[2]"
    - Remember that xpath expressions are strings. It is possible to iterate through a table by parsing integer variables into the xpath: "//table[@id='puzzle']/tbody/tr["+i+"]/td["+j+"]"
- Other Attributes can be useful when a tag does not have an id or a name.
  - The syntax for a tag determined by attribute(s) is: tag[@attribute1='value1' and @attribute2='value2'...]
  - Some web pages have elements with semi-dynamic attributes. For example, an id tag might be generated with a static string concatenated with a dynamic string ("user" + "(unique username)"). In addition to locating elements by fully defined attributes, xpath expressions can locate these elements given partial matches:
    - tag[contains(@attribute,'partial value')]
    - tag[starts_with(@attribute,'partial value')]

- Finally, a parent tag can be determined by its position in a page. This method fragile, much like the absolute path method, and is not recommended.
    - The syntax looks like this: “//tag[4]/div[1]/...”

CSS

CSS selectors are often recommended over xpath. CSS is reputed to be faster (this is not universally true), browsers use CSS (CSS engines are different in each browser, but xpath appears particularly fragile in IE), and the latest browsers optimize the use of CSS selectors. However, xpath allows searching both up and down the DOM tree (CSS only allows parent->child search), handles text recognition, and uses the common language for xml/html parsing.

- Like xpath, CSS can be found using an absolute path
    - The syntax looks like this: driver.findElement(By.cssSelector(“html>body>...(tagName)”));
- We can also use a relative path, using the tag name.
    - driver.findElement(By.cssSelector(“(tagName)”);
- Relative paths will find the first element with the specified tag name. We can specify further by using some regular attributes:
    - driver.findElement(By.cssSelector(“(tagName)[name='cancel']”));
- The following special attributes have more compact syntax:
    - Id: #{elementId}
        - driver.findElement(By.cssSelector("#idOfElement"));
    - class: .{elementClass}
        - driver.findElement(By.cssSelector(".classOfElement")).;
    - Id and tag: elementTag#{elementId}
        - driver.findElement(By.cssSelector("tag#idOfElement")).;
    - class and tag: elementTag.{elementClass}
        - driver.findElement(By.cssSelector("tag.classOfElement")).;
- Nested elements:"elementTag1 elementTag2”
    - <input id=”fname” type=”text” name=”firstName2”>
        - <div id=”parentDiv”>
            - <input id=”fname” type=”text” name=”firstName1”>
    - driver.findElement(By.cssSelector("div input")).sendKeys("Test");
      OR
      driver.findElement(By.cssSelector("div#parentDiv input#fname")).sendKeys("Test");

Part 2: TestNG

What is TestNG?

TestNG is an open-source testing framework that was designed to support the wide range of testing required at the enterprise level. Many of its features, such as supporting dependency and integration testing, were included to address shortcomings of JUnit (which was designed for unit testing). “NG” stands for “next generation”, and developers widely prefer TestNG for its power, flexibility, and ease of use. Complete documentation can be found at http://testng.org/doc/documentation-main.html.

How does TestNG work?

The difference between a TestNG object and a POJO (Plain Old Java Object) is that a TestNG object has at least one annotation supported by TestNG. Developers generally keep their business logic in POJOs and write tests for this logic in separate TestNG objects. TestNG requires a .xml file (generally named testng.xml or build.xml), which contains information about the tests of interest (it could focus on particular suites or test groups, rather than running all tests in the repository).

Unlike standard Java programs, TestNG does not require a main() method. Rather, the entry point for executing TestNG tests is the TestNG class. This is invoked using the testng.xml file in Eclipse. TestNG automatically generates a folder called test-output in the current directory which will contain, among other things, test results in an HTML report.

IMPORTANT NOTE: Running a TestNG test (a TestNG object with at least one @Test annotation) with the Eclipse TestNG plug-in is as simple as right-clicking on the test > Run As... > TestNG Test. This will automatically add all relevant TestNG libraries to the project, and generate a basic testng.xml file. Users who wish to work with TestNG on other platforms will need to add TestNG libraries and create the testng.xml file manually.

Installing TestNG

To use TestNG in Eclipse, first make sure that your Eclipse version is 4.2 or later, and that your Java is 1.7+. Follow the instructions provided at http://testng.org/doc/download.html. This is sufficient for running TestNG with Eclipse. If you want to run TestNG from other platforms (such as Linux), you will need to download the latest TestNG .jar file and configure your machine with a TESTNG_HOME environmental variable, and add that variable to the class path. Instructions for using TestNG outside of Eclipse are out of scope at this time.

TestNG Annotations

TestNG refers to the testng.xml file to determine which tests or groups of tests to run. TestNG determines which tasks to run, and in which order, based on testng.xml flow, but it locates the methods that perform these tasks via annotations. A full list of annotations with attributes can be found here: http://everythingaboutselenium.blogspot.com/2013/07/all-testng-annotations-with-attributes.html. The most common TestNG annotations are as follows:

- **@BeforeSuite/@AfterSuite**
    - Syntax: @BeforeSuite <suiteSetupMethod>, @BeforeSuite(<arguments>) <suiteSetupMethod>; (same for @AfterSuite)
    - Use: Provide instructions for setup, tear-down, reporting and notifications, or anything else desired at the suite level.
    - Details: Add a <test> tag at the beginning (or end) of the suite and include the class containing the relevant <suiteSetupMethod> or <suiteTeardownMethod>. It is not necessary to specify these methods; TestNG will only run @BeforeSuite method(s) at the beginning and @AfterSuite methods at the end.

- **@BeforeGroups/@AfterGroups**
    - Syntax:

        - @Test(groups={“group1name”,...}
        - @BeforeGroups(groups=”<group1name>”) (How do I define list of groups??)
    - Use: To easily include tests that have been flagged as of interest for particular business needs, such as “basicRegression”, “featureX”, etc.

- **@BeforeTest/@AfterTest**
    - Syntax: @BeforeTest <method>, @BeforeTest(<arguments>) <method> (same with @AfterTest)
    - Use: The annotated method will be run before/after each test, as described in the testng.xml file. This is not to be confused with @BeforeMethod/@AfterMethod.

- **@BeforeClass/@AfterClass**
    - Syntax: @BeforeClass <method>, @BeforeClass(<arguments>) <method> (same with @AfterClass)
    - Use: The method annotated with @BeforeClass will be run before the first method in the current class is invoked, and the method annotated with @AfterClass will be run after all @Test methods in the current class have been run.

- **@BeforeMethod/@AfterMethod**
    - Syntax: @BeforeMethod <method>, @BeforeMethod(<arguments>) <method> (same for @AfterMethod)
    - Use: The annotated method will run before/after each method marked with @Test annotation. This is not to be confused with @BeforeTest/@AfterTest.

- **@DataProvider**
    - Syntax: @DataProvider(name=<String literal>), @DataProvider(parallel=<true/false – default is false)
    - Use: The annotated method supplies data to the test method. It must return an Object[][], where each Object[] is the parameter list for one test. This will be illustrated in The Test Base

- **@Parameters**
    - Syntax: @Parameters(“<param1name>”,...)
    - Use: For using parameters defined in the testng.xml file within a given test. XML syntax is <parameter name=”param1name” value=”value” /> . NOTE: Storing parameters in a data file rather than the .xml file is appropriate for data-driven frameworks.

- **@Test**
    - Syntax: @Test, @Test(dataProvider=<String literal matching “name” from @DataProvider>), @Test(<other arguments>)
    - Use: Marks the method as **part of** the test, when the class defining the method is nested inside a test defined in the testng.xml file.

- **@Factory**
    - Syntax: @Factory <method>, @Factory(<arguments>)
    - Use: The annotated method is marked as a factory for generating objects that will be used by TestNG as test classes. The return type required is Object[].

- **@Listeners/@Reporters**
    - Syntax:
    - Use: For generating custom logs and reports

The Testng.xml File

The Eclipse TestNG plugin will automatically generate a simple testng.xml file that assumes one test suite, one test, all TestNG classes included within that test, and no parallel processing. It is not difficult to edit this file to specify one or more tests, suites, or groups. Here is an example of a testng.xls file that was automatically generated by Eclipse upon running a test inside a project :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Suite" parallel="none">
  <test name="Test">
    <classes>
      <class name=".tests.NewTest2"/>
      <class name=".tests.NewTest"/>
      <class name=".tests.LoginTest"/>
      <class name=".tests.TestBase"/>
    </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

Test Suites

A test suite is a collection of tests. Suites have the following attributes:

- name (mandatory)
- verbose – a number from 1-10, as a string (such as “2”), which determines the level of detail of information logged in the console.
- parallel – determines whether multiple threads should be used, defaults to “none”. Valid values also include “tests”, “classes”, …?
- thread-count – number of threads (if and only if parallel mode is enabled), as a string
- annotations – annotation type
- time-out – default timeout value for all tests in suite

Tests

Tests contain a <classes> tag with a list of classes included; a <packages> tag with a list of packages, which in turn contain TestNG classes is also acceptable. Tests might also include a <groups> tag, which can be used to focus a suite to include or exclude methods flagged by a given group name.

Groups

The <groups> tag can contain three different tags:

- <define name=”<metaGroupName>”>
    - list of <include name=”<groupName>” />
    - list of <exclude name=”<groupName>” /> tags.
- <run>
    - list of <include name=”<groupName or metaGroupName>” />

    - list of <exclude name="<groupName or metaGroupName>" />
- <dependencies>
    - list of <group name="<gr1>" depends-on="<gr2> <gr3>" />

Parameters

A list of parameters can be specified within the <test> tag. The syntax of a parameter inside testng.xml is <parameter name="<paramName>" value="<paramValue>" />

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Suite" parallel="none">
  <test name="test1">
    <parameter name="login" value="user1" />
    <parameter name="password" value="pass1" />

    <classes>
      <class name=".tests.LoginTest"/>
    </classes>

  </test> <!-- Test →

  <test name="test2">

    <groups>

      <define name="twoClasses">
        <include name="class1" />
        <include name="class2" />
      </define>

      <run>
        <include name="twoClasses" />
        <exclude name="leaveOut" />
      </run>

    </groups>

      <packages>
        <package name="packageWithClasses" />
      </packages>
  </test>
</suite> <!-- Suite -->
```

Dependencies

Sometimes it is necessary to enforce that tests are performed in a particular order, or to share data and/or state. This can be accomplished by setting one of two attributes within the @Test annotation:

- **dependsOnMethods={"method1",...}** ensures that the method will not be executed until after all of the methods it depends on have been completed
- **dependsOnGroups={"<groupname>.*"}** ensures that the method will not be executed until after all of the methods in <groupname> have been completed. This is more robust to refactoring and code extension.

If test groups are to depend on other test groups, use the <dependencies> tag in the testng.xml file, as described above.

Note that a dependent method will be marked as "SKIP" rather than "FAIL" in the HTML report if any method it depends on fails.

Part 3: Layers of the Framework

The Selenium Layer

A Selenium testing framework should have a layer dedicated to creating and customizing drivers. In a real commercial testing scenario, suites of tests are run in various browsers on one or more environments. The Selenium layer of a testing framework should take configuration information, such as system-specific timeouts and specific browsers, and output an object cast to the interface WebDriver. Selenium is an open-source API subject to ongoing changes, so keeping Selenium-specific code separate from the tests, web page objects, etc. is crucial for efficient maintenance.

Most other WebDriver and WebDriver subclass methods are for:

- Controlling the browser (navigating backward and forward, refreshing the page, adjusting the window size and location, opening and closing windows, and managing cookies) – managed in the Test layer
- Selecting WebElements – managed in the Web Page layer
- Debugging – out of scope at this time

Creating Supported Drivers

Although Selenium can also be used to test mobile applications, the scope of this paper is currently limited to the common web browsers. A class that creates a driver must import the general class WebDriver, plus classes listed under specific drivers. Note that most drivers, except for FirefoxDriver, require some system configuration.

```
import org.openqa.selenium.WebDriver;
```

- **FirefoxDriver**
  - Additional import: `import org.openqa.selenium.firefox.FirefoxDriver;`
  - Additional files: none.
  - Additional system configuration: none.
    - NOTE: FirefoxDriver was developed by the WebDriver/Selenium team, rather than a 3rd party. Because it is fully integrated into the system, no further configuration is necessary.
  - Code to create driver: `WebDriver driver = new FirefoxDriver();`
  - WARNING: FirefoxDriver is being deprecated in favor of MarionetteDriver. It worked in Firefox 46 but fails in Firefox 47, and it is unlikely that it will be fixed.

- **ChromeDriver**
  - Additional import: `import org.openqa.selenium.chrome.ChromeDriver;`
  - Additional files: Download the ChromeDriver.exe from http://code.google.com/p/selenium/downloads/list
  - Additional system configuration: `System.setProperty("webdriver.chrome.driver", “(path to chromedriver.exe)”);`
    - NOTE: ChromeDriver was developed by Google, rather than the Firefox/Selenium team, which is why the additional file chromedriver.exe is necessary.

- Code to create driver: `WebDriver driver = new ChromeDriver();`
- WARNING: ChromeDriver generates an informational message stating “Only local connections are allowed.” The browser will be able to connect to outside web pages, but it will block incoming connections from other websites. It will not affect your testing if your test scripts and ChromeDriver are on the same machine.

- **InternetExplorerDriver**
  - Additional import: `import org.openqa.selenium.ie.InternetExplorerDriver;`
  - Additional files: Download iexploredriver.exe
  - Additional system configuration: `System.setProperty("webdriver.ie.driver", “(path to iexploredriver.exe)”);`
    - NOTE: iexploredriver was developed by a 3rd parth, rather than the Firefox/Selenium team, which is why the additional file is necessary.
  - Code to create driver: `WebDriver driver = new InternetExplorerDriver();`
  - WARNING: IE might require additional settings in order to establish a connection. Details can be found at https://github.com/SeleniumHQ/selenium/wiki/InternetExplorerDriver#Required_Configuration.

- **SafariDriver**
  - Some old instructions for using SafariDriver in Selenium can be found at https://itisatechiesworld.wordpress.com/2015/04/15/steps-to-get-selenium-webdriver-running-on-safari-browser/
  - As of June 2015, Selenium execution on Safari browsers is considered unstable and unreliable.
  - As of October 2015, forum conversations reported that SafariDriver has been hard-coded with a 10-second timeout which can only be addressed by purchasing a $99 developer license from the company that introduced the bug. Safari will remain out of scope for this paper until the fix has been released to the public.

- **Firefox MarionetteDriver**
  - Additional import: `import org.openqa.selenium.firefox.MarionetteDriver;`
  - Additional files: Download the geckodriver.exe
  - Additional system configuration: `System.setProperty("webdriver.gecko.driver", “(path to geckodriver.exe)”);`
    - NOTE: Although Gecko/Marionette is developed by Selenium/Firefox, configuration is necessary at this time. This might change when the driver becomes the default.
  - Code to create driver: `WebDriver driver = new MarionetteDriver();`
  - NOTES: As of 8/5/2016, the driver is “nearly complete”, so expect some instability while it is being integrated. Also note that the official instructions say to add the file location of geckodriver.exe to the system path, but this is not necessary if the system path is used as described.

Customizing the WebDriver Timeout

Tests can fail in a slow environment simply because the driver cannot open before the default timeout limit. This can be mitigated by increasing the timeout. It is appropriate to make this adjustment in the Selenium layer (i.e. the driver generator level) because it is reasonable to assume that a system needing this extension will be universally slow.

Add the following import to your class:

```
import java.util.concurrent.TimeUnit;
```

Adjust the driver timeout, using implicitlyWait() rather than by adding Thread.sleep(), with this code:

```
driver.manage().timeouts().implicitlyWait(t0, TimeUnit.SECONDS);
```

Sample WebDriver Generating Class

```
package selenium;

import java.io.File;
import java.util.concurrent.TimeUnit;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.firefox.MarionetteDriver;
import org.openqa.selenium.ie.InternetExplorerDriver;

public class DriverMaker {

        public DriverMaker(){} // at this time, constructor is very simple

        public WebDriver makeDriver(String browser, String executableFilePath){

                WebDriver driver=null;
                File file=null;
                if(!browser.equalsIgnoreCase("Firefox"))
                        file=new File(executableFilePath);

                try{
                        if(browser.equalsIgnoreCase("Firefox")){
                                driver=new FirefoxDriver();
                                // built-in default for Selenium, no system configuration needed
                                // WARNING: this will fail in versions 47+. Use Marionette instead
                        }
                        else if(browser.equalsIgnoreCase("Chrome")){
                                System.setProperty("webdriver.chrome.driver", file.getAbsolutePath());
                                driver=new ChromeDriver();
                        }
                        else if(browser.equalsIgnoreCase("IE") || browser.equalsIgnoreCase("Internet Explorer")
                                        || browser.equalsIgnoreCase("InternetExplorer")){
                                System.setProperty("webdriver.ie.driver", file.getAbsolutePath());
                                driver=new InternetExplorerDriver();
                        }
                        else if(browser.equalsIgnoreCase("Marionette")){
                                System.setProperty("webdriver.gecko.driver", file.getAbsolutePath());
                                driver=new MarionetteDriver();
                        }
                        else{
                                System.out.println("Error: browser type "+browser+" not supported.");
                        }
                        return driver;
                }
                catch(Exception e){
                        System.out.println("Error: could not make driver with browser "+browser+
                                        " and file "+executableFilePath);
                        return null;
                }
```