Test Automation

Lecture 12 –

Working with Selenium API & Overview of Selenium WebDriver & Working with WebDriver

IT Learning &
Outsourcing Center

Lector: Milen Strahinski
Skype: strahinski
E-mail: milen.strahinski@pragmatic.bg
Facebook: http://www.facebook.com/LamerMan
LinkedIn: http://www.linkedin.com/pub/milen-strahinski/a/553/615

www.pragmatic.bg

Summary - overall

- Learn the history of Selenium WebDriver
- Architecture
- Run a test with Firefox
  - Working with Firefox profiles
- Run a test with Google Chrome or Chromium
  - Updating the capabilities of the browser
- Run a test with Opera
  - Working with Opera Profiles
- Run a test with Internet Explorer
  - Working with InternetExplorerDriver
- Checking text, attributes and CSS values
- Mouse & Keyboard Events
- Execution of a JavaScript code
- Screenshots
- Automating dropdowns, radio buttons, checkboxes, browser window
- Controlling windows registry with WebDriver

Learn the history of Selenium WebDriver (part 1)

- Jason Huggins – created Selenium IDE and Selenium RC(Remote Control) – this initially solved the problem of getting the browser to do user interactions.

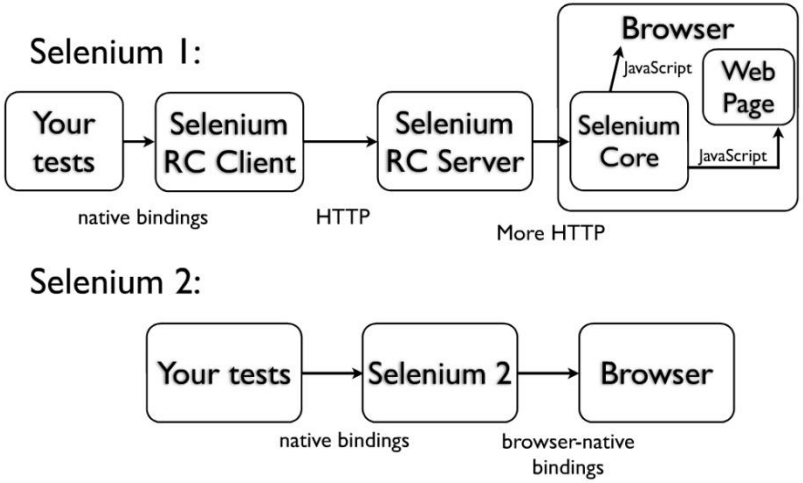
- Patrick Lightbody and Paul Hammant thought that there must be a better way to drive their tests and in a way that they could use their favorite development language. They created Selenium Remote Control using Java as a web server that would proxy traffic. It would inject Selenium onto the page and then it would be used in a similar manner as to what it was in the three column manner. This also creates more of a procedural style of development.

Learn the history of Selenium WebDriver (part 2)

- Simon Stewart, having hit a number of these issues, wanted to try a different approach to driving the browser. While working for ThoughtWorks, he started working on the WebDriver project. It started originally as a way to drive HTMLUnit and Internet Explorer but having learnt lessons from Selenium RC, Simon was able to design the API to fit in with the way most developers think. Developers have been doing Object Orientated development for a while, so moving away from the procedural style of Selenium RC was a welcome change to developers. For those interested I suggest reading Simon Stewart's article on Selenium design at http://www.aosabook.org/en/selenium.html

Architecture

- The WebDriver architecture does not follow the same approach as Selenium RC, which was written purely in JavaScript for all the browser automation.



- WebDriver on the other hand tries to control the browser from outside the browser. It uses accessibility API to drive the browser

Working with WebDriver – preliminary points

- IE Driver Executable: http://selenium-release.storage.googleapis.com/index.html
- Chrome Driver Executable: http://chromedriver.storage.googleapis.com/index.html
- Opera Driver Executable: https://github.com/operasoftware/operadriver/downloads
- Firefox Driver does not require a download as it is bundled with the Java client bindings
- Please make sure that you have all the necessary browsers installed respectively for the drivers above.

Our First Test Example


```
import org.junit.*;
import org.openqa.selenium.*;
import org.openqa.selenium.firefox.*;
import java.io.File;
import java.util.Dictionary;

public class OurFirstTestClass {
        WebDriver driver;

        @Before
        public void setUp() {
            driver = //we will update this part with each section
            driver.get("http://pragmatic.bg/automation/");
        }

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

        @Test
        public void testExamples() {
            // We will put examples in here
        }
}
```

JUnit @Annotations

Annotation	Description
@Test public void method()	The annotation @Test identifies that a method is a test method.
@Before public void method()	This method is executed before each test. This method can prepare the test environment (e.g. read input data, initialize the class).
@After public void method()	This method is executed after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeClass public static void method()	This method is executed once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
@AfterClass public static void method()	This method is executed once, after all tests have been finished. This can be used to perform clean-up activities, for example to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
@Ignore	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.
@Test (expected = Exception.class)	Fails, if the method does not throw the named exception.
@Test(timeout=100)	Fails, if the method takes longer than 100 milliseconds.

Working with FirefoxDriver (part 1)

- FirefoxDriver is the easiest driver to use, since everything that we need to use is all bundled with the Java client bindings.

1. Update the *setUp()* method to load the

```
FirefoxDriver driver = new FirefoxDriver();
```

2. Now we need to find an element. In this section we will find the one with the ID *nextBid*:

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

3. Now we need to type into that element:

```
element.sendKeys("100");
```

Working with FirefoxDriver (part 2)

4. Run your test and it should look like the following:

```
public class OurFirstTestClass {
        WebDriver driver;

        @Before
        public void setUp() {
            driver = new FirefoxDriver();
            driver.get("http://pragmatic.bg/automation/example4");
        }

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

        @Test
        public void testExamples() {
            WebElement element = driver.findElement(By.id("nextBid"));
            element.sendKeys("100");
        }
}
```

Working with FirefoxDriver Profiles (part 1)

- There are times where we need to update the preferences within Firefox.
  - This could be to switch on parts of Firefox that are disabled while they are in development
  - or you want to get more information from the browser while your tests are running

```
FirefoxProfile profile = new FirefoxProfile();
profile.setPreference("browser.startup.homepage","http://pragmatic.bg");
driver = new FirefoxDriver(profile);
```

Working with FirefoxDriver Profiles (part 2)


```
public class OurSecondExampleClass {
        WebDriver driver;

        @Before
        public void setUp(){
            FirefoxProfile profile = new FirefoxProfile();
            profile.setPreference("browser.startup.homepage",
           "http://www.pragmatic.bg/automation/example4");
            driver = new FirefoxDriver(profile);
        }

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

        @Test
        public void testExamples(){
            WebElement element = driver.findElement(By.id("nextBid"));
            element.sendKeys("100");
        }
}
```

Working with FirefoxDriver Profiles (part 3)

- If you had installed Firefox in a different place, you would have had to instantiate the *FirefoxBinary* class with details of it:

```
FirefoxBinary binary = new FirefoxBinary("/path/to/binary");
driver = new FirefoxDriver(binary);
```

- If you need to update both the Firefox Profile and the Firefox Binary, you can simply pass both of them through the constructor as follows:

```
FirefoxBinary binary = new FirefoxBinary("/path/to/binary");
FirefoxProfile profile = new FirefoxProfile();
profile.setPreference("browser.startup.homepage",
      “http://pragmatic.bg/automation/example4.html");
driver = new FirefoxDriver(binary, profile);
```

Installing Firefox Add-on


```
public class InstallingFirefoxAddon {
        WebDriver driver;

        @Before
        public void setUp() {
            FirefoxProfile profile = new FirefoxProfile();
            profile.addExtension("firebug.xpi");
            driver = new FirefoxDriver(profile);

            driver.get("http://www.pragmatic.bg/automation/example4");
        }

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

        @Test
        public void testExamples() {
            WebElement element = driver.findElement(By.id("nextBid"));
            element.sendKeys("100");
        }
}
```

Working with ChromeDriver (part 1)

- Imagine that you wanted to work with Google Chrome to get an attribute of an element on the page. To do this we will need to instantiate a ChromeDriver. Let's see an example.

```
System.setProperty("webdriver.chrome.driver",
"C:\\path\\to\\chromedriver.exe");
driver = new ChromeDriver();
WebElement element =
   driver.findElement(By.id("selectLoad"));
element.getAttribute("value");
```

Working with ChromeDriver (part 2)


```
public class ChromeDriverExample {
        WebDriver driver;

        @Before
        public void setUp() {
            System.setProperty("webdriver.chrome.driver",
"C:\\Users\\Strahinski\\Desktop\\Automated Testing Course\\Lectures\\Lecture 12
- Working with Selenium API & Overview of Selenium
WebDriver\\chromedriver.exe");
            driver = new ChromeDriver();
            driver.get("http://www.pragmatic.bg/automation/example4");
        }

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

        @Test
        public void testExamples() {
            WebElement element = driver.findElement(By.id("selectLoad"));
            String value = element.getAttribute("value");
            Assert.assertEquals("Click to load the select below", value);
        }
}
```

ChromeOptions (part 1)

- Google Chrome or Chromium doesn't really have a profile that users can update in the same sense as Firefox. It does however have a mechanism that allows us to set certain options that Chrome will try and use. We can also tell it to install Chromium extensions, which are like Firefox add-ons, into the browser so we can enhance the experience.

ChromeOptions (part 2)

- Refer to ChromeDriverExample.java

- We have just seen how we can inject options that we want Chrome or Chromium to start with. If we needed to pass in the arguments that we could start the browser with or if we needed to tell ChromeDriver, we can use addArguments(). This allows us to do many things to the browser. We can see a definitive list at
- http://src.chromium.org/viewvc/chrome/trunk/src/chrome/common/chrome_switches.cc?view=markup
- http://www.askvg.com/how-to-access-hidden-secret-advanced-configuration-pages-in-mozilla-firefox-google-chrome-and-opera-web-browsers/

ChromeOptions (part 3)

- If you have a Chrome Extension, a file with a .crx extension, you will need to use the addExtension() method as you would in FirefoxDriver. The following snippet will show an example:

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

Working with OperaDriver (part 1)

- Opera Software, the company that creates Opera, has created their own project to support Selenium WebDriver. Since not every web browser will act the same with the sites that we create, it is a good idea to make sure we can test our applications with OperaDriver.

- Imagine that you want to test your web application that uses geolocation in the browser, when it cannot use geolocation. All location-based applications need to support this if you were to get a user who is worried about privacy on certain machines.

Working with OperaDriver (part 2)


```
public class OperaDriverExample {
Webdriver driver;
@Before
public void setUp() {
    OperaProfile profile = new OperaProfile();
    profile.preferences().set("Geolocation", "Enable geolocation",
    false);
    driver = new OperaDriver();
    driver.get("http://pragmatic.bg/automation/");
    }

@Test
public void testExamples() {
    WebElement element =
    driver.findElement(By.partialLinkText("Chapter4"));
    element.click();
    driver.manage().timeouts().implicitlyWait(1, TimeUnit.SECONDS);
    // Assert that we only have 1 link
    Assert.assertEquals(1,
    driver.findElements(By.linkText("Index")).size());
}
. . . . . .
```

Working with InternetExplorerDriver (part 1)

- Internet Explorer is the most used browser in the world followed by Firefox and Google Chrome so getting IEDriver working is going to be a high priority. The current version IEDriver supports IE6 through to IE9 so you will be able to test your websites work on old browsers right up to the latest modern version of the browser.

Working with InternetExplorerDriver (part 2)


```
public class InternetExplorerDriverExample {
WebDriver driver;

@Before
public void setUp(){
    System.setProperty("webdriver.ie.driver",
                                "C:\\path\\to\\IEDriverServerx32.exe");
    driver = new InternetExplorerDriver();
    driver.get("http://pragmatic.bg/automation/example4.html");
}

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

@Test
public void testExamples(){
    WebElement element = driver.findElement(By.id("bid"));
    Assert.assertEquals("50", element.getText());
}
```

Checking an element's text (part 1)

- While testing a web application, we need to verify that elements are displaying correct values or text on the page.
- Sometimes, we need to retrieve text or value from an element into a variable at runtime and later use it at some other place in the test flow.
- Using the WebElement class' getText() method.

Checking an element's text (part 2)


```
public class ElementTests {
//Open http://dl.dropbox.com/u/55228056/DoubleClickDemo.html
@Test
public void testElementText()
{
    //Get the message Element
    WebElement message = driver.findElement(By.id("message"));

    //Get the message elements text
    String messageText = message.getText();

    //Verify message element's text displays "Click on me and my color
    will change"
    assertEquals("Click on me and my color will change", messageText);

    //Get the area Element
    WebElement area = driver.findElement(By.id("area"));

    //Verify area element's text displays "Div's Text\nSpan's Text"
    assertEquals("Div's Text\nSpan's Text",area.getText());
}
```

Checking an element's text (part 3)

- There is more:
  - assertTrue(messageText.contains("color"));
  - assertTrue(messageText.startsWith("Click on"));
  - assertTrue(messageText.endsWith("will change"));

Checking an element's attribute values

- Developers configure various attributes of elements displayed on the web page during design or at runtime to control the behavior or style of elements when they are displayed in the browser. For example, the <input> element can be set to read-only by setting the *readonly* attribute.

```
@Test  //that's in class ElementTests.java
public void testElementAttribute()
{
    //open http://dl.dropbox.com/u/55228056/DoubleClickDemo.html
    WebElement message = driver.findElement(By.id("message"));
    assertEquals("justify",message.getAttribute("align"));
}
```

Checking an element's CSS values

- There may be tests that need to verify that correct styles have been applied to the elements. This can be done using WebElement class's *getCSSValue()* method, which returns the value of a specified style attribute.

```
@Test
public void testElementStyle()
{
    //open http://dl.dropbox.com/u/55228056/DoubleClickDemo.html

    WebElement message = driver.findElement(By.id("message"));
    String width = message.getCssValue("width");
    assertEquals("150px",width);
}
```

Using Advanced User Interactions API for mouse and keyboard events (part 1)

- The Selenium WebDriver's Advanced User Interactions API allows us to perform operations from keyboard events and simple mouse events to complex events such as dragging-and-dropping, holding a key and then performing mouse operations by using the *Actions* class, and building a complex chain of events exactly like a user doing these manually

Using Advanced User Interactions API for mouse and keyboard events (part 2)


```
@Test          //open class RowSelectionTests.java
public void testRowSelectionUsingControlKey() {
driver.get("http://jsbin.com/ofupam");
WebElement gish = new WebDriverWait(driver,
10).until(ExpectedConditions.presenceOfElementLocated(By.xpath("//table[@id='fruits']")));

List<WebElement> allRows = driver.findElements(By.xpath("//table[@id='fruits']/tbody/tr"));

//Select second and fourth row from Table using Control Key.
//Row Index start at 0
Actions builder = new Actions(driver);

builder.keyDown(Keys.CONTROL)
.click(allRows.get(0))
.click(allRows.get(1))
.click(allRows.get(3))
.keyUp(Keys.CONTROL);

Action selectMultiple = builder.build();
selectMultiple.perform();

//Verify Selected Rows in Table are exactly how we expect
int numberOfSelectedRows=
driver.findElements(By.xpath("//table[@id='fruits']/tbody/tr[contains(@class,'ui-
selected')]")).size();

assertEquals(3, numberOfSelectedRows);
}
```

Performing double-click on an element (part 1)

- There will be elements in a web application that need double-click events fired for performing some actions. For example, double-clicking on a row of a table will launch a new window. The Advanced User Interaction API provides a method to perform double-click.

Performing double-click on an element (part 2)


```
@Test //open class DoubleClickTest.java
public void testDoubleClick() throws Exception
{
    WebDriver driver = new FirefoxDriver();
    driver.get("http://dl.dropbox.com/u/55228056/DoubleClickDemo.html");

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

    //Verify color is Blue
    assertEquals("rgba(0, 0, 255, 1)", message.getCssValue("background-
    color").toString());

    Actions builder = new Actions(driver);
    builder.doubleClick(message).build().perform();

    //Verify Color is Yellow
    assertEquals("rgba(255, 255, 0, 1)",message.getCssValue("background-
    color").toString());

    driver.close();
}
```

Performing drag-and-drop operations (part 1)

- Selenium WebDriver implements Selenium RC's dragAndDrop command using Actions class. As seen in earlier recipes the Actions class supports advanced user interactions such as firing various mouse and keyboard events. We can build simple or complex chains of events using this class.

Performing drag-and-drop operations (part 2)


```
@Test    //open class DragDropTest.java
public void testDragDrop() {

    driver.get("http://dl.dropbox.com/u/55228056/DragDropDemo.html");

    WebElement source = driver.findElement(By.id("draggable"));
    WebElement target = driver.findElement(By.id("droppable"));

    Actions builder = new Actions(driver);
    builder.dragAndDrop(source, target) .perform();
    try
    {
       assertEquals("Dropped!", target.getText());
    } catch (Error e) {
       verificationErrors.append(e.toString());
    }
}
```

Executing JavaScript code (part 1)

- Selenium WebDriver API provides the ability to execute JavaScript code with the browser window. This is a very useful feature where tests need to interact with the page using JavaScript. Using this API, client-side JavaScript code can also be tested using Selenium WebDriver. Selenium WebDriver provides a JavascriptExecutor interface that can be used to execute arbitrary JavaScript code within the context of the browser.

Executing JavaScript code (part 2)


```
@Test    //open class ExecuteJavaScript.java
public void testJavaScriptCalls() throws Exception
{
    System.setProperty("webdriver.chrome.driver", "C:\\path
    to\\chromedriver.exe");
    WebDriver driver = new ChromeDriver();
    driver.get("http://www.google.com");

    JavascriptExecutor js = (JavascriptExecutor) driver;

    String title = (String) js.executeScript("return
    document.title");
    assertEquals("Google", title);

    long links = (Long) js.executeScript("var links =
    document.getElementsByTagName('A'); return links.length");
    assertEquals(35, links);

    driver.quit();

}
```

Capturing screenshots with Selenium WebDriver (part 1)

- Selenium WebDriver provides the *TakesScreenshot* interface for capturing a screenshot of a web page. This helps in test runs, showing exactly happened when an exception or error occurred during execution, and so on. We can also capture screenshots during verification of element state, values displayed, or state after an action is completed.

- Capturing screenshots also helps in verification of layouts, field alignments, and so on where we compare screenshots taken during test execution with baseline images.

Capturing screenshots with Selenium WebDriver (part 2)


```
@Test //open class ScreenshotTests.java
public void testTakesScreenshot()
{
  try {
   File scrFile =
        ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
   FileUtils.copyFile(scrFile, new File("c:\\tmp\\main_page.png"));
  } catch (Exception e) {
        e.printStackTrace();
  }
}
```

Capturing screenshots with Selenium WebDriver (part 3)

- Take a screenshot of a specific element (this will require a helper method, cuz it's not supported by default by Selenium TakeScreenshot class)

```
@Test
public void testElementScreenshot(){

   WebElement pmoabsdiv = driver.findElement(By.className("pmoabs"));
   try {
     FileUtils.copyFile(ScreenshotTests.captureElementBitmap(pmoabsdiv),
new File("c:\\tmp\\div.png"));
   } catch (Exception e) {
        e.printStackTrace();
   }
 }
```

- The helper captureElementBitmap on the next slide

Capturing screenshots with Selenium WebDriver (part 4)


```
public static File captureElementBitmap(WebElement element) throws Exception {
  //Get the WrapsDriver of the WebElement
  WrapsDriver wrapsDriver = (WrapsDriver) element;

  //Get the entire Screenshot from the driver of passed WebElement
  File screen = ((TakesScreenshot)
wrapsDriver.getWrappedDriver()).getScreenshotAs(OutputType.FILE);

  //Create an instance of Buffered Image from captured screenshot
  BufferedImage img = ImageIO.read(screen);

  // Get the Width and Height of the WebElement using
  int width = element.getSize().getWidth();
  int height = element.getSize().getHeight();

  //Create a rectangle using Width and Height
  Rectangle rect = new Rectangle(width, height);

  //Get the Location of WebElement in a Point. ....
  //This will provide X & Y co-ordinates of the WebElement
  Point p =  element.getLocation();
  //Create image by for element using its location and size.
  //This will give image data specific to the WebElement
  BufferedImage dest = img.getSubimage(p.getX(), p.getY(), rect.width, rect.height);
  //Write back the image data for element in File object
  ImageIO.write(dest, "png", screen);
  //Return the File object containing image data
  return screen;
}
```

Maximizing the browser window

- Selenium RC's windowMaximize() command was missing in Selenium WebDriver. However starting from release 2.21, Selenium WebDriver supports maximizing the browser window.

```
@Test  //open class WindowMaximize.java
public void testRowSelectionUsingControlKey() throws Exception {

    WebDriver driver = new FirefoxDriver();

    driver.get("http://www.google.com");
    driver.manage().window().maximize();
    Thread.sleep(1000);
    driver.quit();
}
```

Automating dropdowns and lists (part 1)

- Selenium WebDriver supports testing Dropdown and List controls using a special Select class instead of the WebElement class.

- The Select class provides various methods and properties to interact with dropdowns and lists created with the HTML <select> element.

Automating dropdowns and lists (part 2)


```
@Test
public void testDropdown() {
 //open class SelectTest.java
    Select make = new Select(driver.findElement(By.name("make")));

    assertFalse(make.isMultiple());
    assertEquals(4, make.getOptions().size());

    List<String> exp_options = Arrays.asList(new String[]{"BMW", "Mercedes",
"Audi","Honda"});
    List<String> act_options = new ArrayList<String>();

    for(WebElement option : make.getOptions())
       act_options.add(option.getText());

    assertArrayEquals(exp_options.toArray(),act_options.toArray());

    make.selectByVisibleText("Honda");
    assertEquals("Honda", make.getFirstSelectedOption().getText());

    make.selectByValue("audi");
    assertEquals("Audi", make.getFirstSelectedOption().getText());

    //or we can select an option in Dropdown using index
    make.selectByIndex(0);
    assertEquals("BMW", make.getFirstSelectedOption().getText());
}
```

Automating dropdowns and lists (part 3)

- Create another test for a List control which has multi-selection enabled. The test will perform some basic checks and then call methods to select multiple options in a list. The test will verify the selection and then deselect the options by calling various deselection methods, namely by visible text, by value, and by index, respectively.

Automating dropdowns and lists (part 4)


```
@Test
public void testMultipleSelectList() {
    Select color = new Select(driver.findElement(By.name("color")));
    assertTrue(color.isMultiple());
    assertEquals(5, color.getOptions().size());
    color.selectByVisibleText("Black");
    color.selectByVisibleText("Red");
    color.selectByVisibleText("Silver");
    assertEquals(3,color.getAllSelectedOptions().size());

    List<String> exp_sel_options = Arrays.asList(new String[]{"Black", "Red",
"Silver"});
    List<String> act_sel_options = new ArrayList<String>();

    for(WebElement option : color.getAllSelectedOptions())
    act_sel_options.add(option.getText());

    assertArrayEquals(exp_sel_options.toArray(),act_sel_options.toArray());
    color.deselectByVisibleText("Silver");
    assertEquals(2,color.getAllSelectedOptions().size());
    color.deselectByValue("rd");
    assertEquals(1,color.getAllSelectedOptions().size());
    color.deselectByIndex(0);
    assertEquals(0,color.getAllSelectedOptions().size());
}
```

Automating radio buttons and radio groups (part 1)

- Selenium WebDriver supports Radio Button and Radio Group controls using the WebElement class. We can select and deselect the radio buttons using the click() method of the WebElement class and check whether a radio button is selected or deselected using the isSelected() method.

Automating radio buttons and radio groups (part 2)


```
@Test
public void testRadioButton(){   //Open class RadioButtonTest.java
    WebElement petrol = driver.findElement(By.xpath("//input[@value='Petrol']"));

//Check if its already selected? Otherwise select the RadioButton by calling click()
    if (!petrol.isSelected())
      petrol.click();
    //Verify Radio Button is selected
    assertTrue(petrol.isSelected());
    //We can also get all the Radio buttons from a Radio Group in a list
    //using findElements() method along with Radio Group identifier
    List<WebElement> fuel_type = driver.findElements(By.name("fuel_type"));
    for (WebElement type : fuel_type)
    {
      //Search for Diesel Radio Button in the Radio Group and select it
      if(type.getAttribute("value").equals("Diesel"))
      {
        if(!type.isSelected())
            type.click();

      assertTrue(type.isSelected());
      break;
      }
    }
}
```

Automating checkboxes (part 1)

- Selenium WebDriver supports Checkbox control using the WebElement class. We can select or deselect a checkbox using the click() method of the WebElement class and check whether a checkbox is selected or deselected using the isSelected() method.

Automating checkboxes (part 2)

```
@Test
public void testCheckBox() {
 //open class CheckBoxTest.java
 //Get the Checkbox as WebElement using it's value attribute
 WebElement airbags =
        driver.findElement(By.xpath("//input[@value='Airbags']"));

 //Check if its already selected? otherwise select the Checkbox
 //by calling click() method
 if (!airbags.isSelected())
    airbags.click();

 //Verify Checkbox is Selected
 assertTrue(airbags.isSelected());

 //Check Checkbox if selected? If yes, deselect it
 //by calling click() method
 if (airbags.isSelected())
    airbags.click();

 //Verify Checkbox is Deselected
 assertFalse(airbags.isSelected());
}
```

Controlling a Windows process (part 1)

- Selenium WebDriver Java bindings provide the WindowsUtils class with methods to interact with the Windows operating system. During test runs, there might be a need to close open instances of browser windows or processes at the beginning of the test. By using the WindowsUtils class, we can control the process and perform tasks, such as killing an open process, and so on.

Controlling a Windows process (part 2)

- Now lets modify some of our tests and Make it work?

```
@Before
public void setUp()
{
    WindowsUtils.tryToKillByName("firefox.exe");
    driver = new FirefoxDriver();
    driver.get("http://www.google.com");
    driver.manage().window().maximize();
}
```

Reading a Windows registry value from Selenium WebDriver (part 1)

- The WindowsUtils class provides various methods to interact with the registry on the Windows operating system. While running tests on the Windows operating system and Internet Explorer, the test might need to read and modify IE settings or there may be a need to get some settings related to the web server or database from the registry in tests. The WindowsUtils class comes handy in these situations.

Reading a Windows registry value from Selenium WebDriver (part 2)


```
@Test  //open class WindowsRegistry
public void testReadRegistry()
{
    String osname =
WindowsUtils.readStringRegistryValue("HKEY_LOCAL_MACHINE\\SOFTWARE\\Micro
soft\\Windows NT\\CurrentVersion\\ProductName");

    System.out.println(osname);
}
```

- If the data type is Integer then we can use the readIntegerRegistryValue() method and for Boolean, we can use readBooleanRegistryValue().

Modifying a Windows registry value from Selenium WebDriver (part 1)

- The WindowsUtils class also provides methods to update existing Windows registry values or create new registry keys and values. Similar to reading registry values, WindowsUtils class provides multiple methods to modify keys and values.

Modifying a Windows registry value from Selenium WebDriver (part 2)


```
@Test
public void testWriteRegistry()
{
WindowsUtils.writeStringRegistryValue("HKEY_CURRENT_USER\\SOFTWARE\\Selen
ium\\SeleniumVersion", "2.33");

assertEquals("2.33",WindowsUtils.readStringRegistryValue("HKEY_CURRENT_US
ER\\SOFTWARE\\Selenium\\SeleniumVersion"));
}
```

- The WindowsUtils class provides multiple methods based on value type such a writeIntegerRegistryValue() for integer values and writeBooleanRegistryValue() for Boolean data type.

Questions?