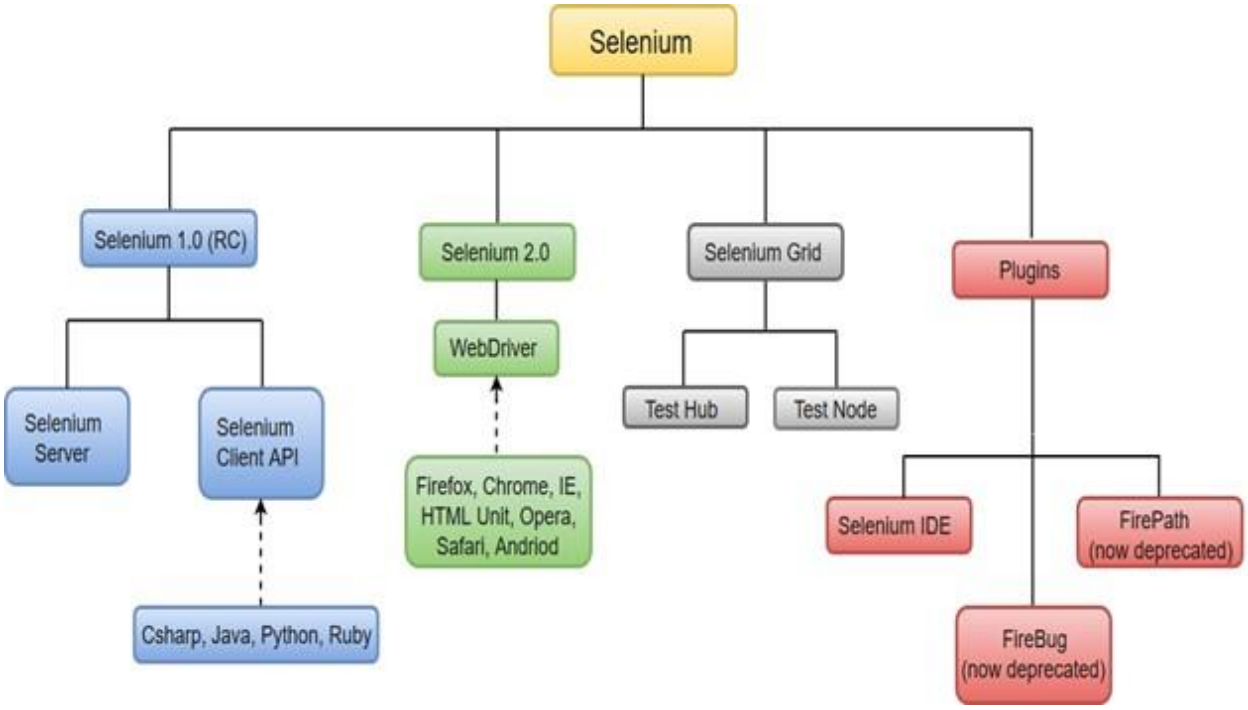
Unit 7 Selenium Driver

Selenium WebDriver is the most important component of Selenium Tool's Suite. The latest release "Selenium 2.0" is integrated with WebDriver API which provides a simpler and more concise programming interface.

The following image will give you a fair understanding of Selenium components and the Test Automation Tools.



Selenium WebDriver was first introduced as a part of Selenium v2.0. The initial version of Selenium i.e Selenium v1 consisted of only IDE, RC and Grid. However, with the release of Selenium v3, RC has been deprecated and moved to legacy package.

In WebDriver, test scripts can be developed using any of the supported programming languages and can be run directly in most modern web browsers. Languages supported by WebDriver include C#, Java, Perl, PHP, Python and Ruby.

Selenium WebDriver performs much faster as compared to Selenium RC because it makes direct calls to the web browsers. RC on the other hand needs an RC server to interact with the browser.

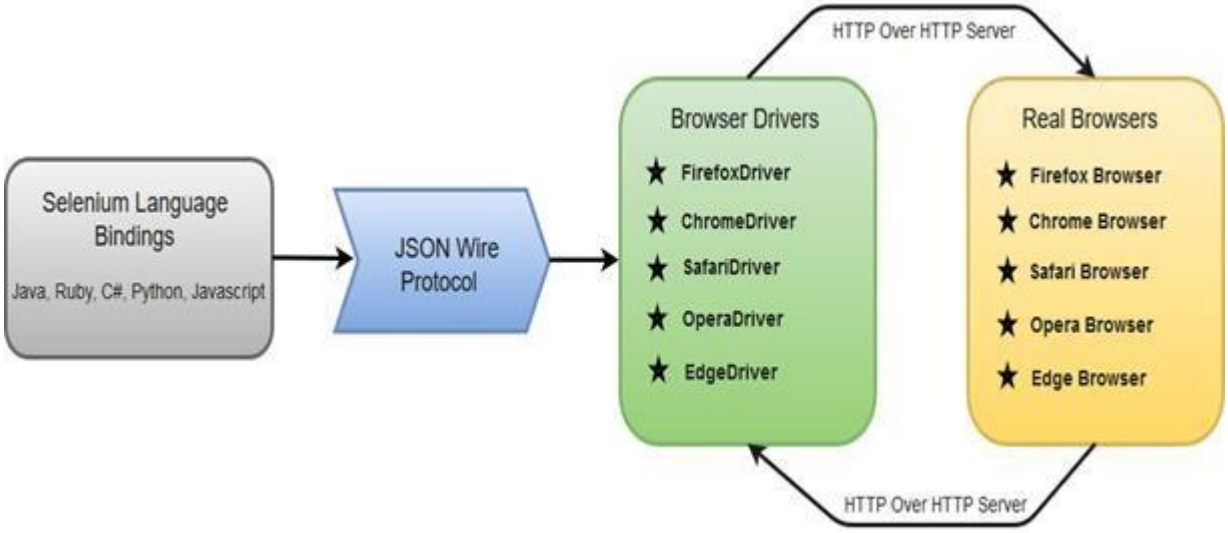
WebDriver has a built-in implementation of Firefox driver (Gecko Driver). For other browsers, you need to plug-in their browser specific drivers to communicate and run the test. Most commonly used WebDriver's include:

- Google Chrome Driver
- Internet Explorer Driver
- Opera Driver
- Safari Driver
- HTML Unit Driver (a special headless driver)

Selenium WebDriver- Architecture

Selenium WebDriver API provides communication facility between languages and browsers.

The following image shows the architectural representation of Selenium WebDriver.



There are four basic components of WebDriver Architecture:

- Selenium Language Bindings
- JSON Wire Protocol
- Browser Drivers
- Real Browsers

Selenium Language Bindings / Selenium Client Libraries

Selenium developers have built language bindings/Selenium Client Libraries in order to support multiple languages. For instance, if you want to use the browser driver in java, use the java bindings. All the supported language bindings can be downloaded from the official website (https://www.seleniumhq.org/download/#client-drivers) of Selenium.

JSON Wire Protocol

JSON (JavaScript Object Notation) is an open standard for exchanging data on web. It supports data structures like object and array. So, it is easy to write and read data from JSON. To learn more about JSON, visit https://www.javatpoint.com/json-tutorial

JSON Wire Protocol provides a transport mechanism to transfer data between a server and a client. JSON Wire Protocol serves as an industry standard for various REST web services. To learn more about Web Services, visit https://www.javatpoint.com/web-services-tutorial

Browser Drivers

Selenium uses drivers, specific to each browser in order to establish a secure connection with the browser without revealing the internal logic of browser's functionality. The browser driver is also specific to the language used for automation such as Java, C#, etc. When we execute a test script using WebDriver, the following operations are performed internally.

- HTTP request is generated and sent to the browser driver for each Selenium command.
- The driver receives the HTTP request through HTTP server.
- HTTP Server decides all the steps to perform instructions which are executed on browser.
- Execution status is sent back to HTTP Server which is subsequently sent back to automation script.

Browsers

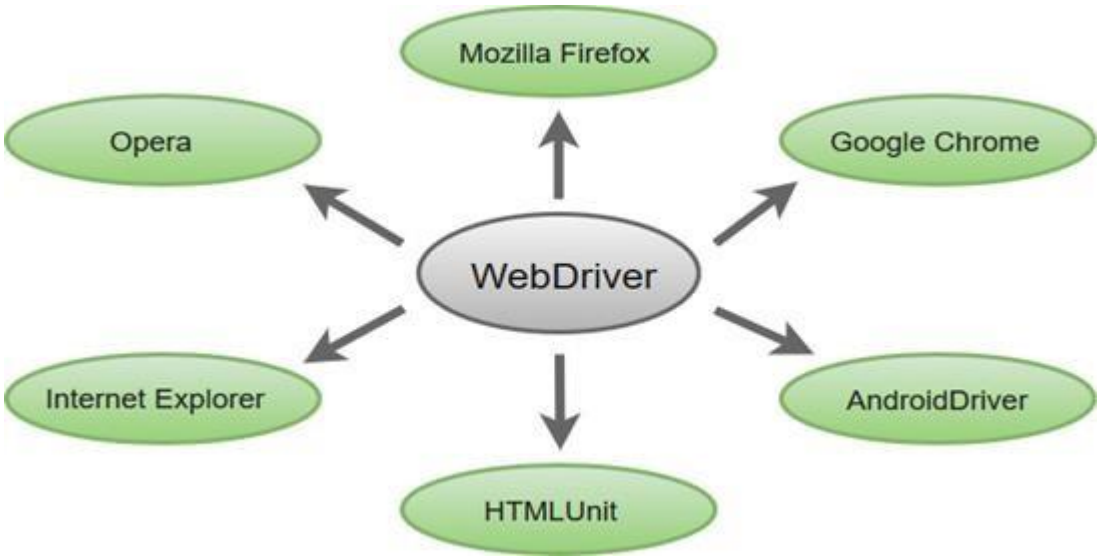
Browsers supported by Selenium WebDriver:

- Internet Explorer
- Mozilla Firefox
- Google Chrome
- Safari

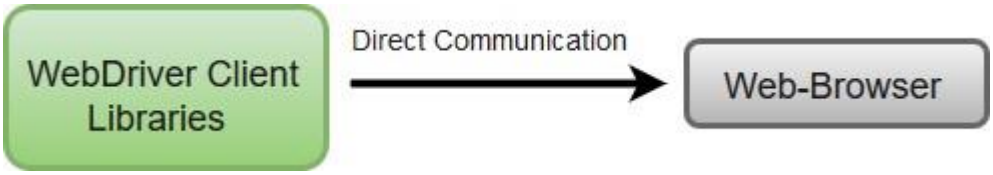
Selenium WebDriver- Features

Some of the most important features of Selenium WebDriver are:

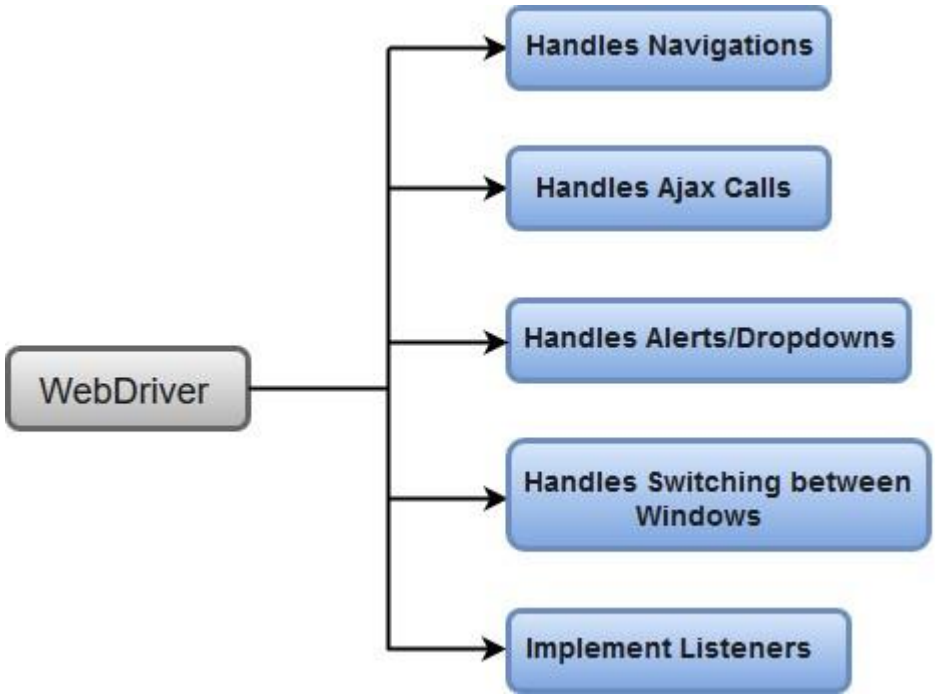
- **Multiple Browser Support**: Selenium WebDriver supports a diverse range of web browsers such as Firefox, Chrome, Internet Explorer, Opera and many more. It also supports some of the non-conventional or rare browsers like HTMLUnit.



- **Multiple Languages Support**: WebDriver also supports most of the commonly used programming languages like Java, C#, JavaScript, PHP, Ruby, Pearl and Python. Thus, the user can choose any one of the supported programming language based on his/her competency and start building the test scripts.
- **Speed**: WebDriver performs faster as compared to other tools of Selenium Suite. Unlike RC, it doesn't require any intermediate server to communicate with the browser; rather the tool directly communicates with the browser.



- **Simple Commands**: Most of the commands used in Selenium WebDriver are easy to implement. For instance, to launch a browser in WebDriver following commands are used:
  WebDriver driver = new FirefoxDriver(); (Firefox browser)
  WebDriver driver = new ChromeDriver(); (Chrome browser)
  WebDriver driver = new InternetExplorerDriver(); (Internet Explorer browser)
- **WebDriver- Methods and Classes**: WebDriver provides multiple solutions to cope with some potential challenges in automation testing. WebDriver also allows testers to deal with complex types of web elements such as checkboxes, dropdowns and alerts through dynamic finders.



Selenium WebDriver Vs Selenium RC

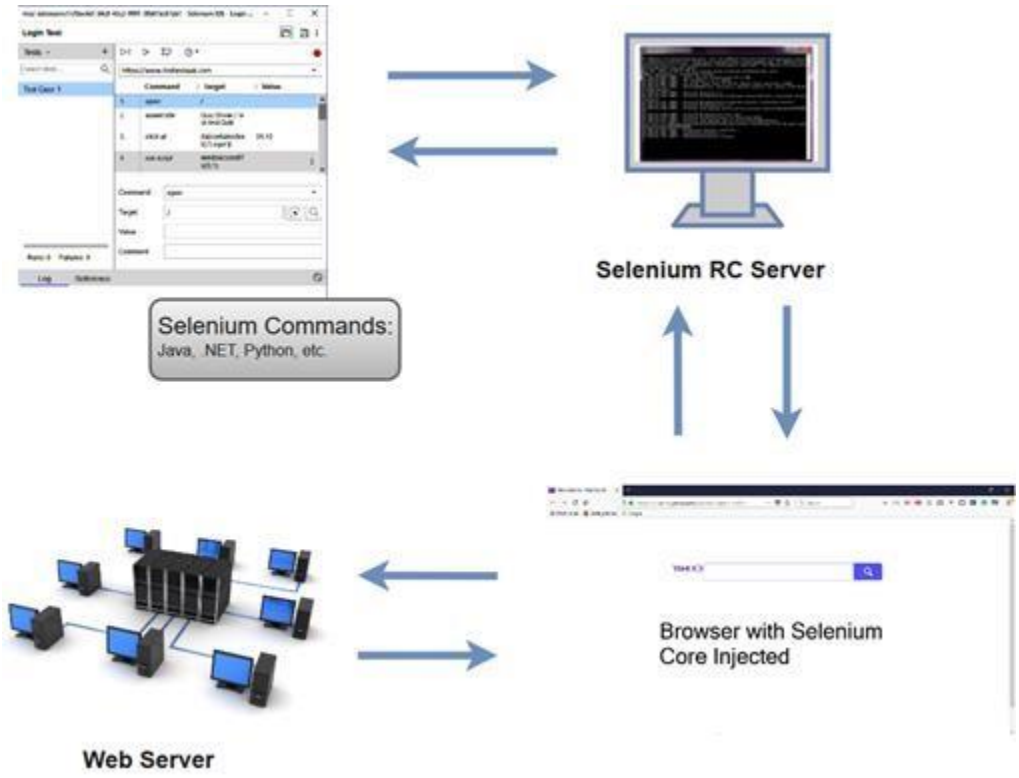
Selenium RC had a lot of limitations which eventually led to the development of Selenium WebDriver.

See the major differences between Selenium RC and Selenium WebDriver:

1. Architecture

The architecture of Selenium RC is complicated because it uses an intermediate RC Server to communicate with the browser. The RC Server is installed initially before running the test scripts and acts as mediator between your Selenium commands and your browser. When we execute a test script in Selenium RC, the following operations are performed internally.

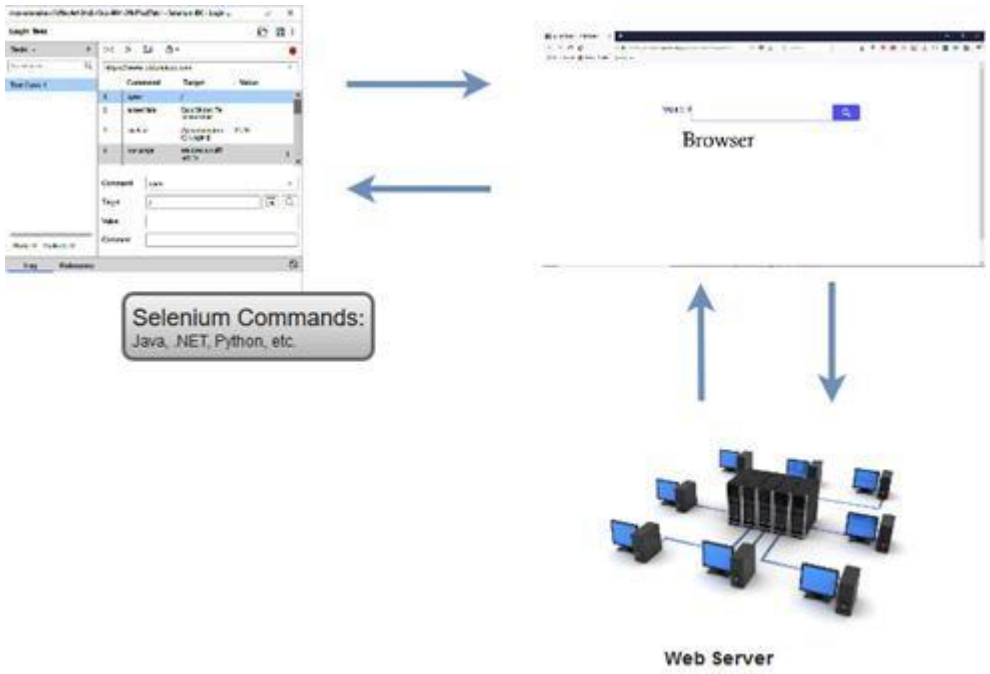
- The server injects a JavaScript program known as **Selenium Core** into the browser.
- Subsequently, Selenium Core will start receiving the instructions (Selenium commands) from the RC Server.
- When all the instructions are received, Selenium Core will execute them as **JavaScript commands**.
- These JavaScript commands act as instructions to the browser.
- The browser will execute all of the instructions provided by Selenium Core and returns an overall summary to the Server. This overall summary acts as the final result which is displayed on the user screen.



The architecture of Selenium WebDriver is simpler as compared to Selenium RC.

The browser is controlled directly from OS (Operating System) level. The basic requirements to run a test script on WebDriver are:

- An IDE (Integrated Development Environment) with any of the supported programming language like Java, C#, etc. • A Browser to execute the instructions generated by the test script.



2. Speed

Selenium WebDriver performs faster than Selenium RC because it interacts directly with the browser without using any external proxy server. Selenium RC, on the other hand uses an intermediate RC Server to communicate with the browser.

Execution of test scripts takes more time in Selenium RC than WebDriver, since it uses JavaScript commands as instructions to the browser.

3. Object Oriented

Selenium WebDriver is purely object oriented API, whereas Selenium RC is less object oriented API.

WebDriver is entirely based on object oriented programming languages like Java, C#, etc.

4. Testing Mobile Applications

Selenium WebDriver supports OS (Operating System) for mobile applications like iOS, windows mobile and android. On the other hand, Selenium RC doesn't support testing of mobile applications.

5. Browser Support

Selenium WebDriver also supports headless HTMLUnit browser (Invisible Browser).

Note: HTMLUnit is an invisible browser which facilitates faster execution of tests because it involves no time in waiting for page elements to load.

Selenium RC doesn't support the headless HTMLUnit browser as it needs a real browser to work with.

Selenium WebDriver- Installation

Selenium WebDriver installation process is completed in four basic steps:

1. Download and Install Java 8 or higher version.
2. Download and configure Eclipse or any Java IDE of your choice.
3. Download Selenium WebDriver Java Client
4. Configure Selenium WebDriver

1. Download and Install Java

We assume that you have already installed Java 8 or above on your machine and successfully configured the environment variables required to run and compile java programs.

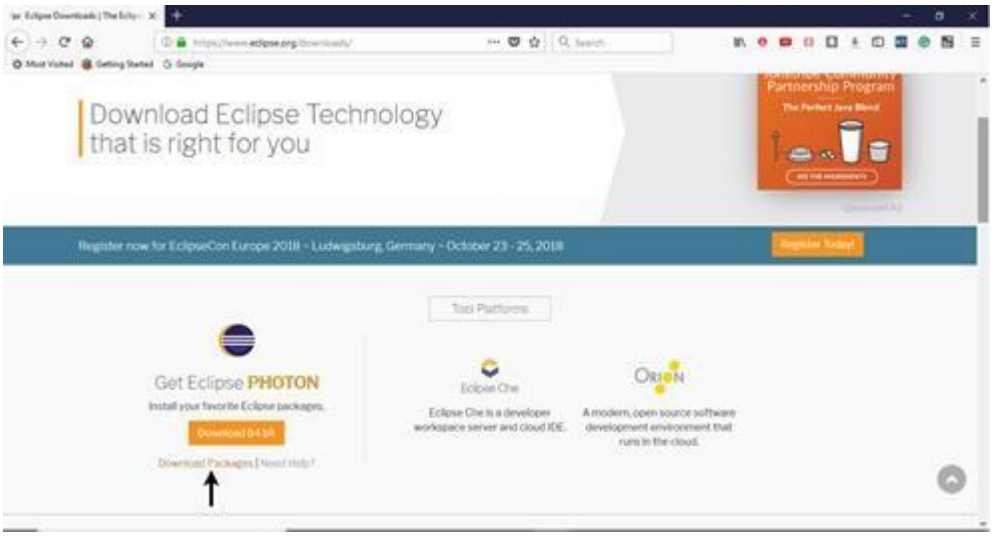
Note: you'll need to have Java 8 installed to use Selenium 3.

However, you can download the latest version of Java Development Kit (JDK) from the link given below. http://www.oracle.com/technetwork/java/javase/downloads/index.html

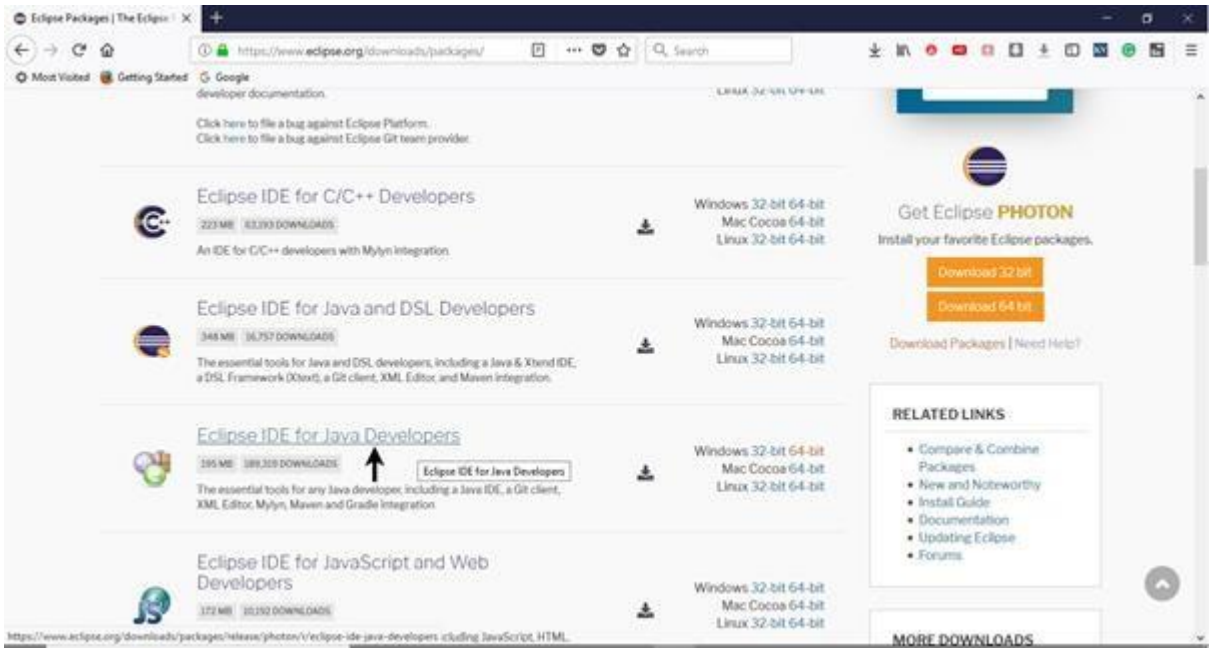
Once you have downloaded and installed the latest version of Java, you need to set path or configure the environment variables in your system. Refer the link given below to understand how we can set path and configure environment variables in Java. https://www.javatpoint.com/how-to-set-path-in-java

2. Download and Configure Eclipse IDE

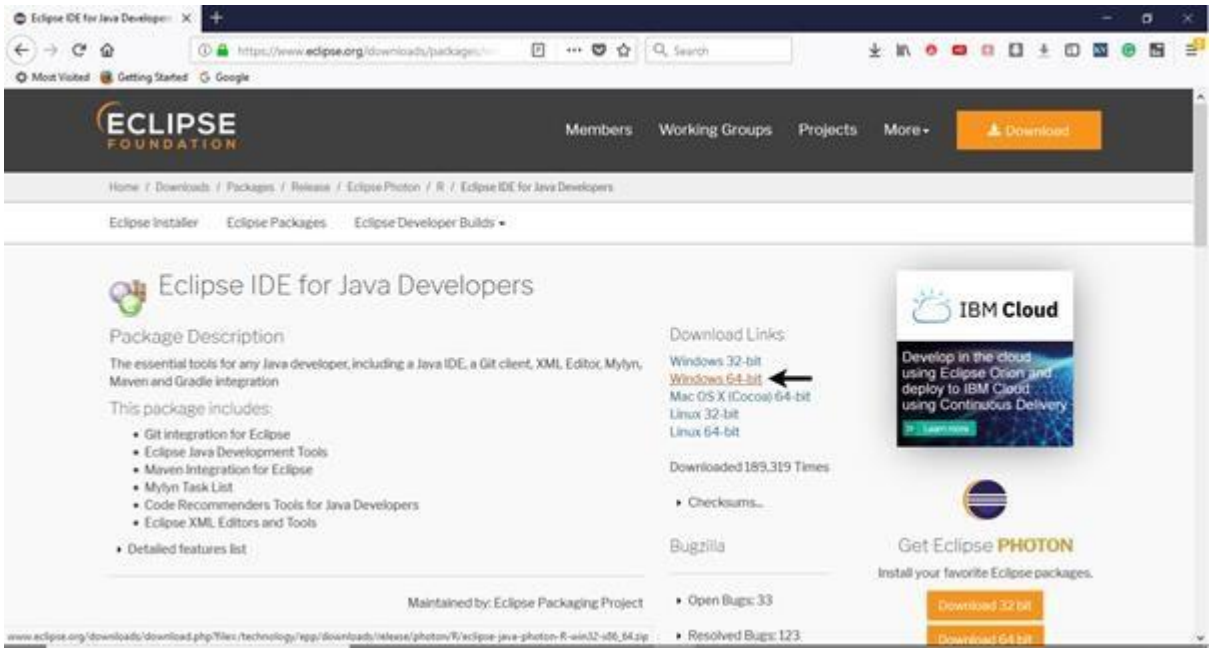
- Open URL: https://www.eclipse.org/downloads/ .
- Click on the "Download Packages" link (you can also download the IDE directly from the "downloads page" of Eclipse official website, but we will recommend you to navigate through the download packages section and get "Eclipse IDE for Java Developers").



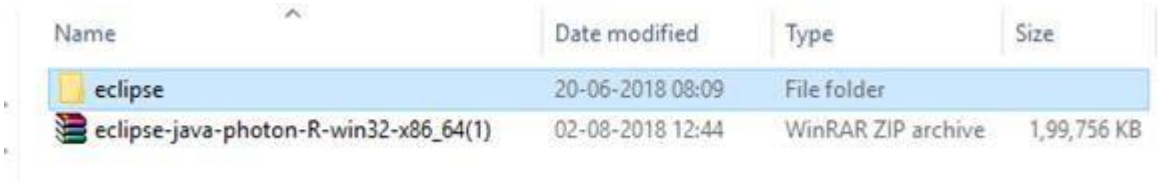
- It will redirect you to the "Download Packages" section. Scroll down through the webpage and click on "Eclipse IDE for Java Developers".



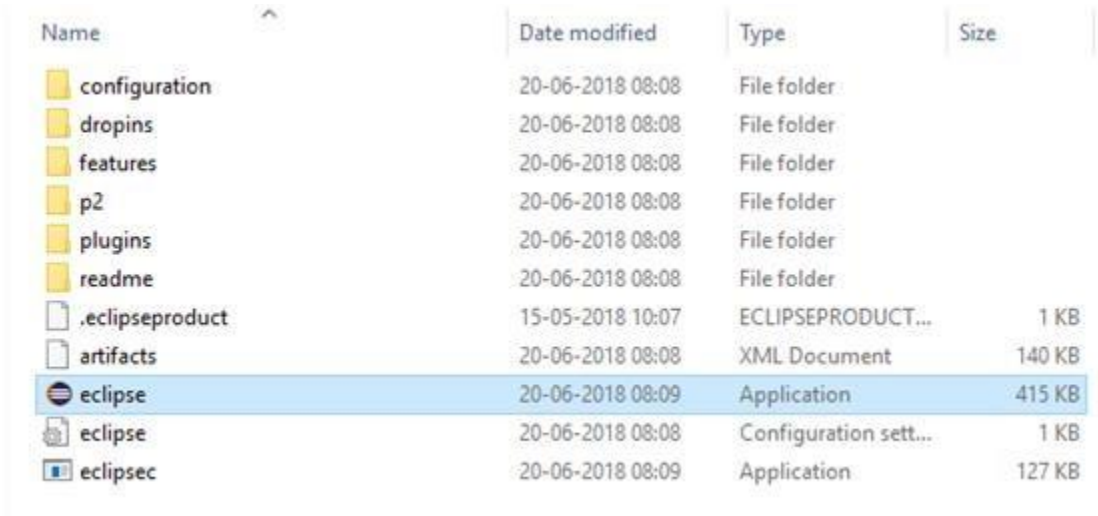
- Go to the Download Links section and click on "Windows 64-bit". You can also select other options to download based on the operating system you are currently working on.



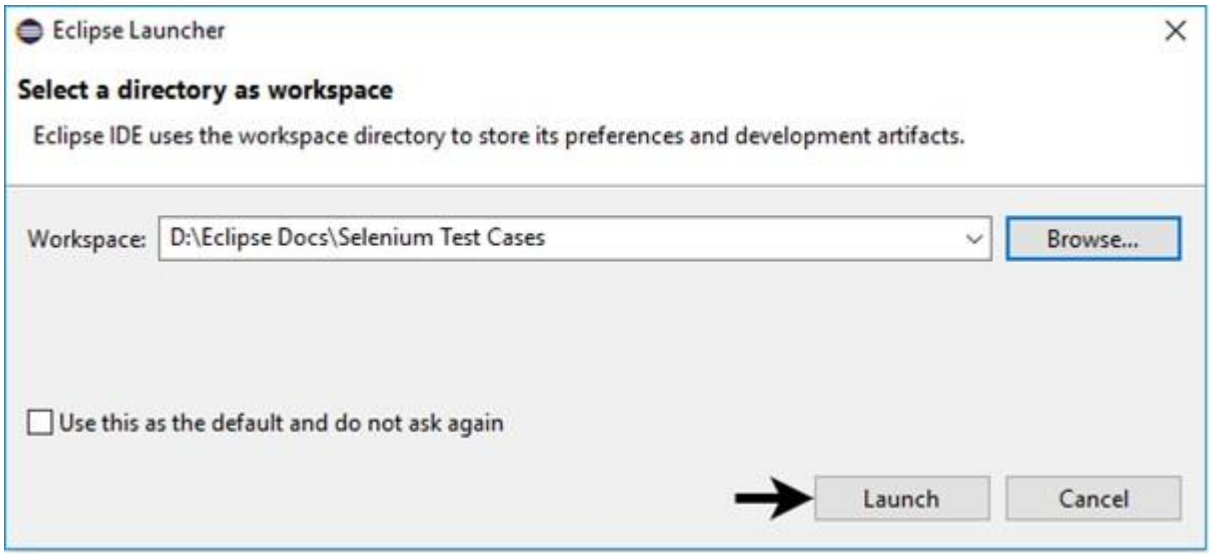
- The downloaded file would be in zipped format. Unpack the contents in a convenient directory.



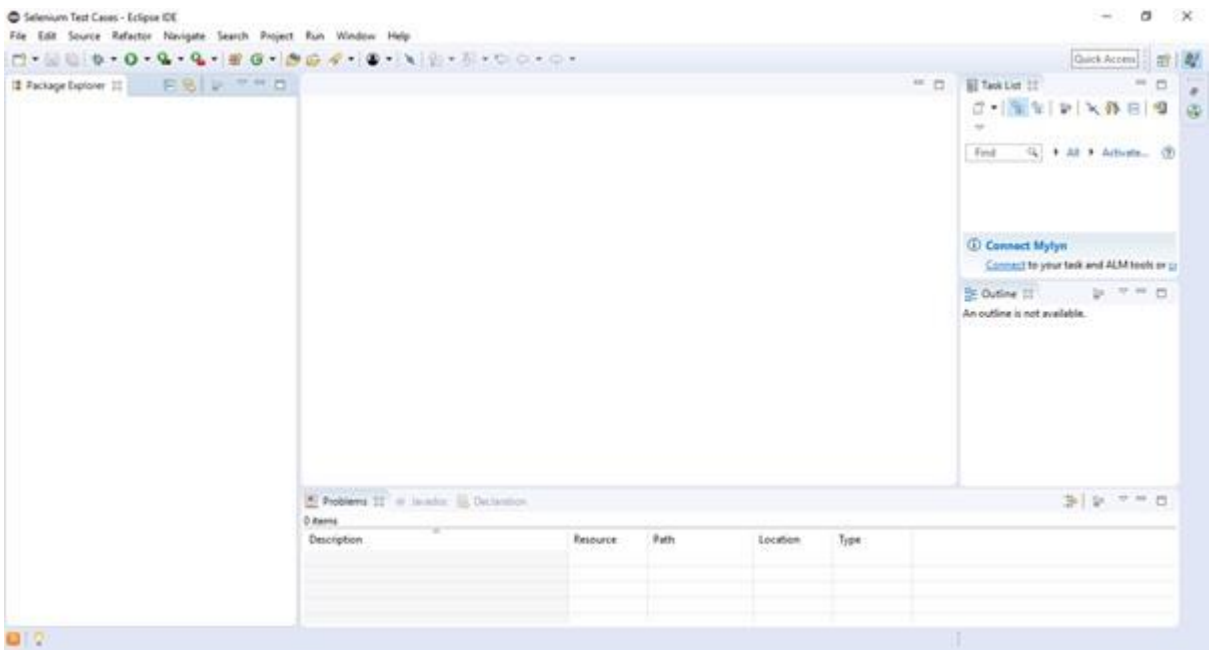
- Double click on "eclipse" (.exe file).



- To configure the workspace, select a convenient directory where you want to keep all of your Selenium trails and click on Launch button.



- It will launch the default interface of Eclipse IDE.



3. Download Selenium WebDriver Java Client

- Open URL: https://docs.seleniumhq.org/download/
  It will redirect you to the "downloads page" of Selenium official website.
- Scroll down through the web page and locate **Selenium Client & WebDriver Language Bindings**. • Click on the "Download" link of Java Client Driver as shown in the image given below.

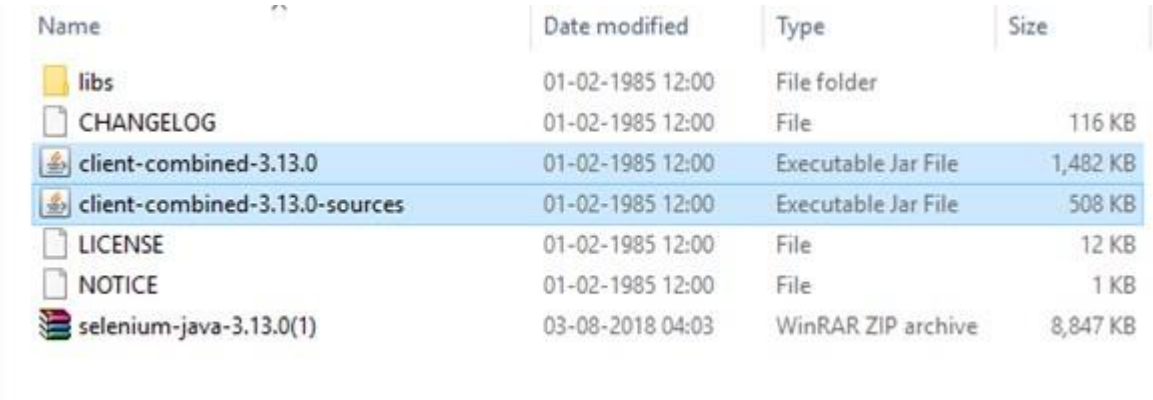
Selenium Client & WebDriver Language Bindings

In order to create scripts that interact with the Selenium Server (Selenium RC, Selenium Remote WebDriver) or create local Selenium WebDriver scripts, you need to make use of language-specific client drivers. These languages include both 1.x and 2.x style clients.

While language bindings for other languages exist, these are the core ones that are supported by the main project hosted on GitHub.

Language	Client Version	Release Date			
Java	3.13.0	2018-06-25	Download	Change log	Javadoc
C#	3.13.0	2018-06-25	Download	Change log	API docs
Ruby	3.13.1	2018-07-20	Download	Change log	API docs
Python	3.13.0	2018-06-25	Download	Change log	API docs
Javascript (Node)	4.0.0-alpha.1	2018-01-13	Download	Change log	API docs

- The downloaded file would be in zipped format. Unpack the contents in a convenient directory. It contains the essential jar files required to configure Selenium WebDriver in Eclipse IDE.

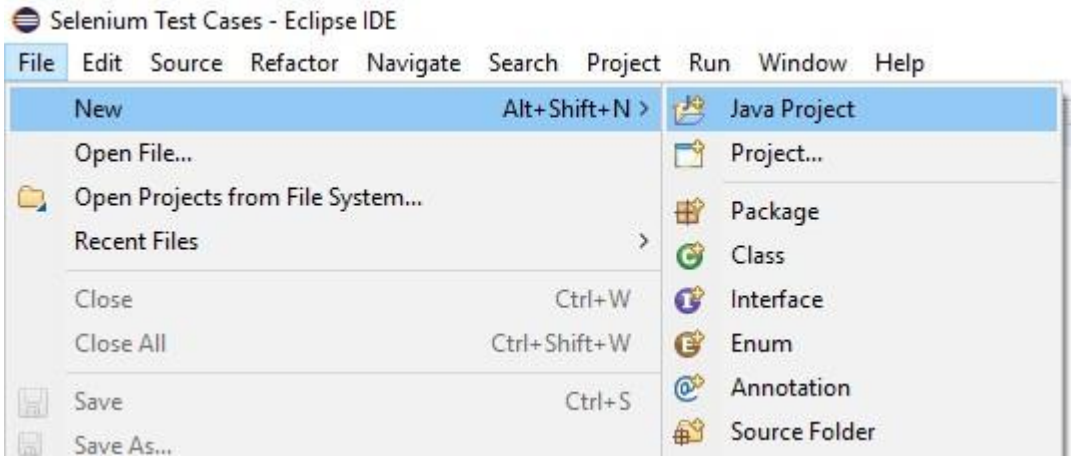


Name	Date modified	Type	Size
libs	01-02-1985 12:00	File folder	
CHANGELOG	01-02-1985 12:00	File	116 KB
client-combined-3.13.0	01-02-1985 12:00	Executable Jar File	1,482 KB
client-combined-3.13.0-sources	01-02-1985 12:00	Executable Jar File	508 KB
LICENSE	01-02-1985 12:00	File	12 KB
NOTICE	01-02-1985 12:00	File	1 KB
selenium-java-3.13.0(1)	03-08-2018 04:03	WinRAR ZIP archive	8,847 KB

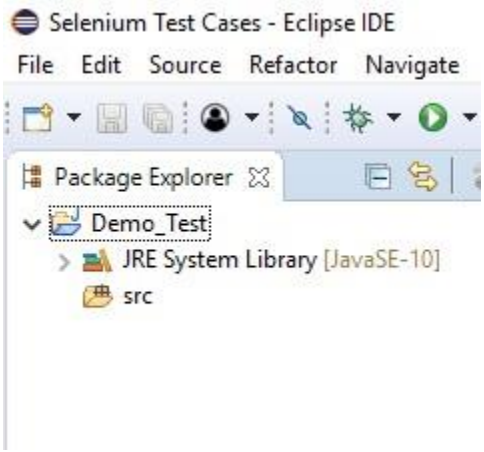
4. Configure Selenium WebDriver

Now we will configure our Eclipse IDE with Selenium WebDriver. In simple terms, we will create a new Java Project in Eclipse and load all the essential jar files in order to create Selenium Test Scripts.

- Launch Eclipse IDE • Create a new Java Project from **File > New > Java Project**.

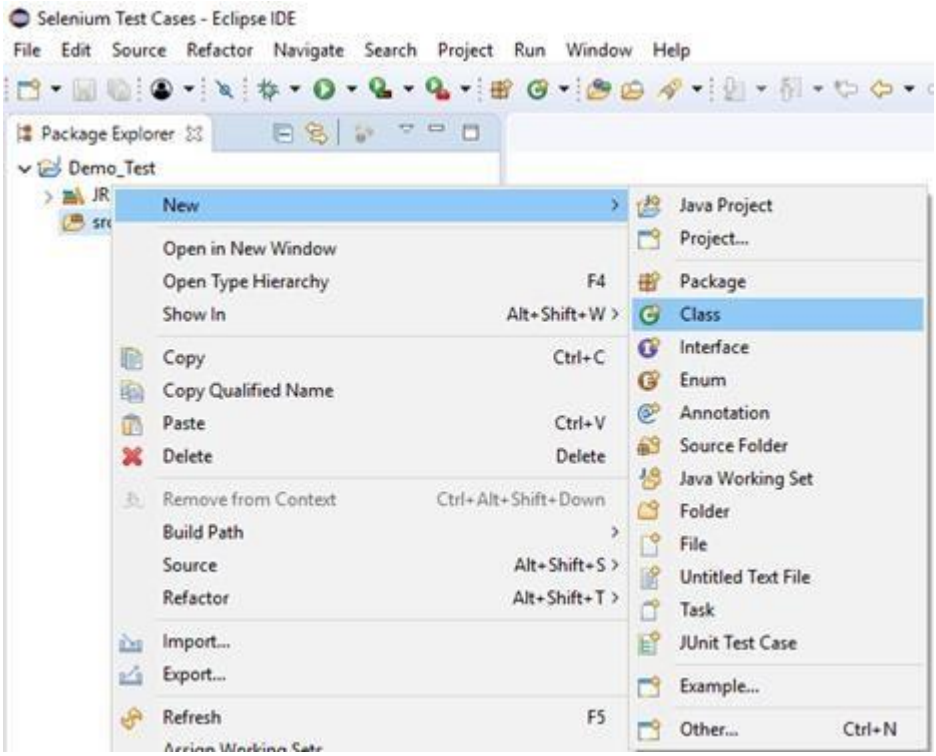


- Give your Project name as "Demo_Test", leave the other fields unaltered and click on "Finish" button. • It will create a new Java project with the following directories.

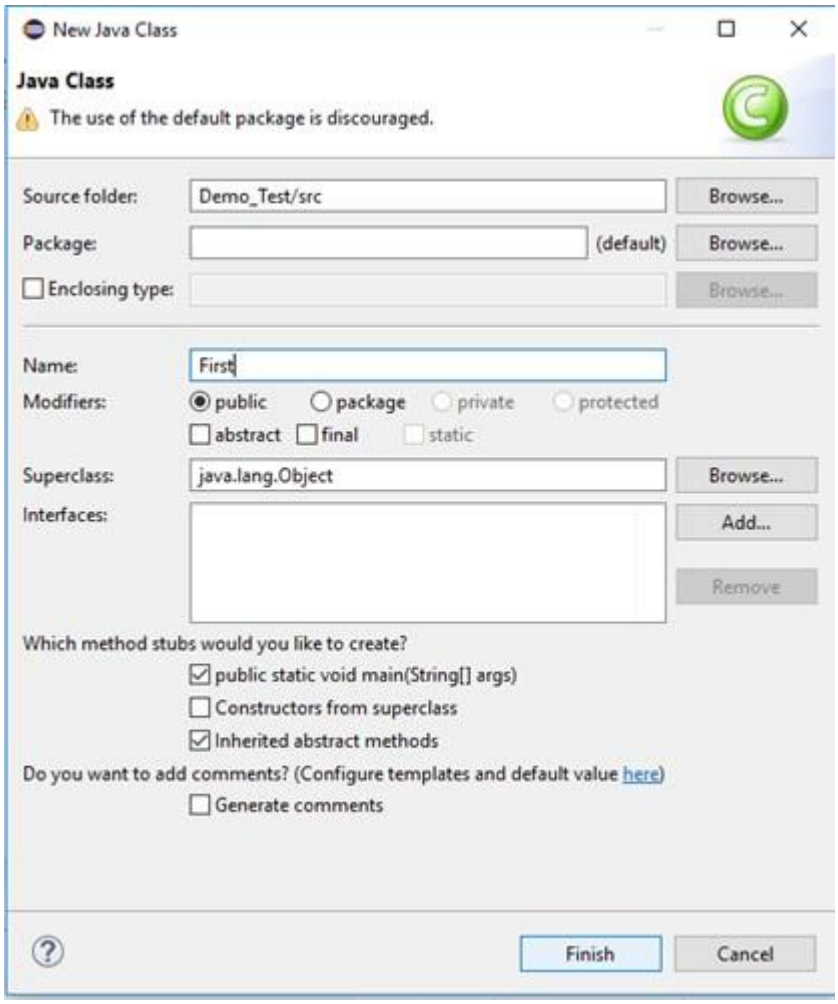


Note: Selenium Test Scripts are always written in ".class" file in Java. Here the project "Demo_Test" act as a Test Suite that may contain one or more Selenium test cases/test scripts.

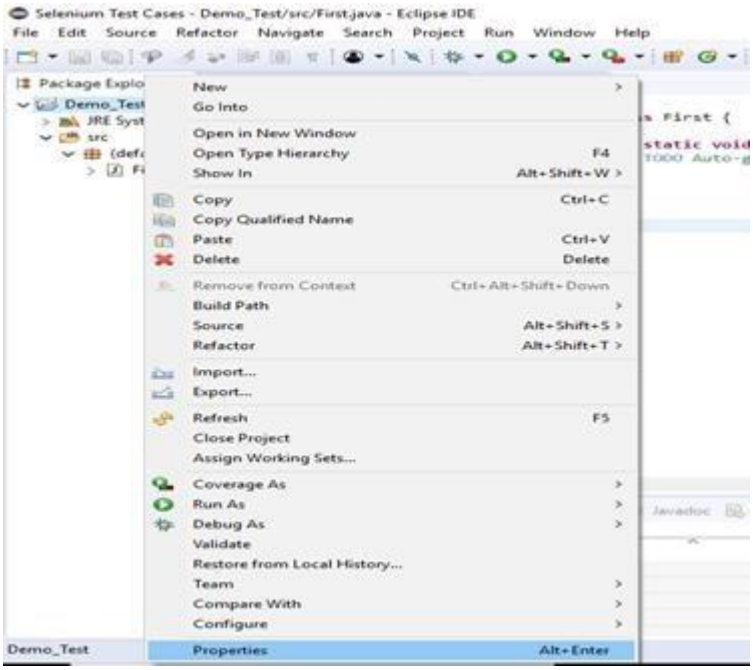
- Right click on the "src" folder and create a new Class File from **New > Class**.



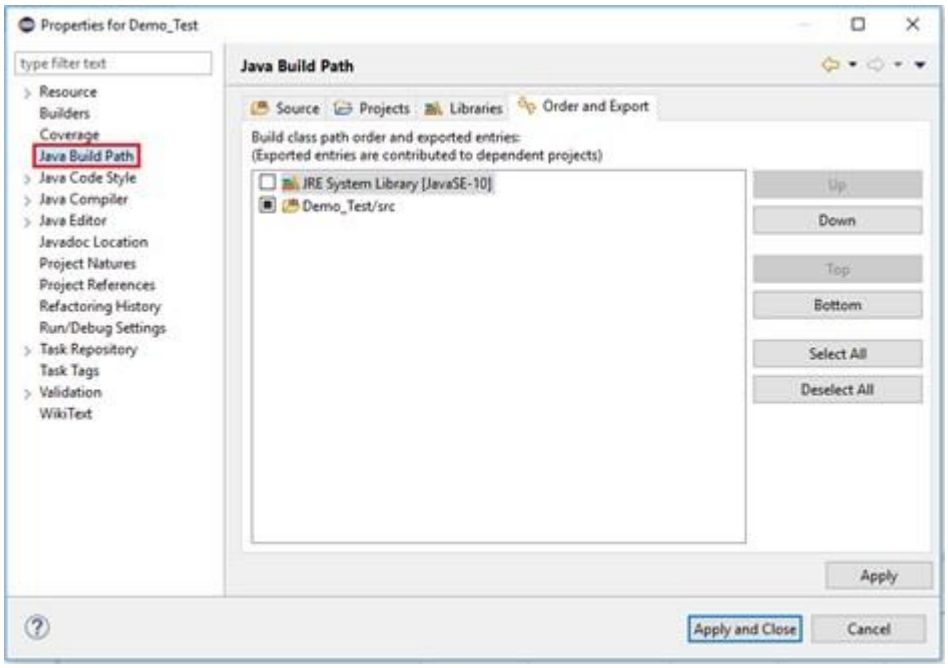
- Give your Class name as "First" and click on "Finish" button.



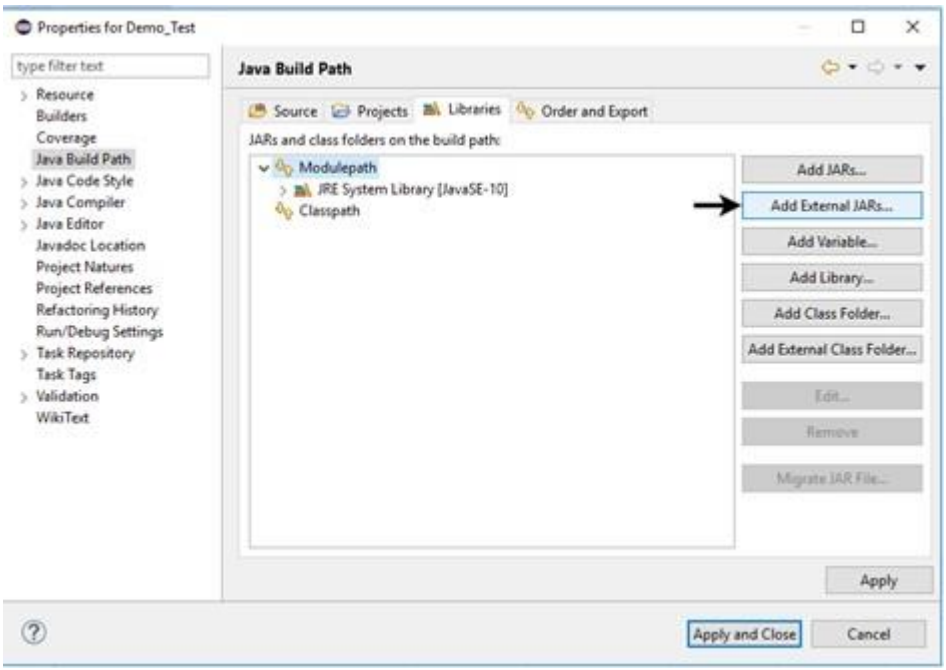
Now, we will add the Selenium jar files in our Test Suite (Demo_Test). •
Right click on "Demo_Test" folder and select Properties.



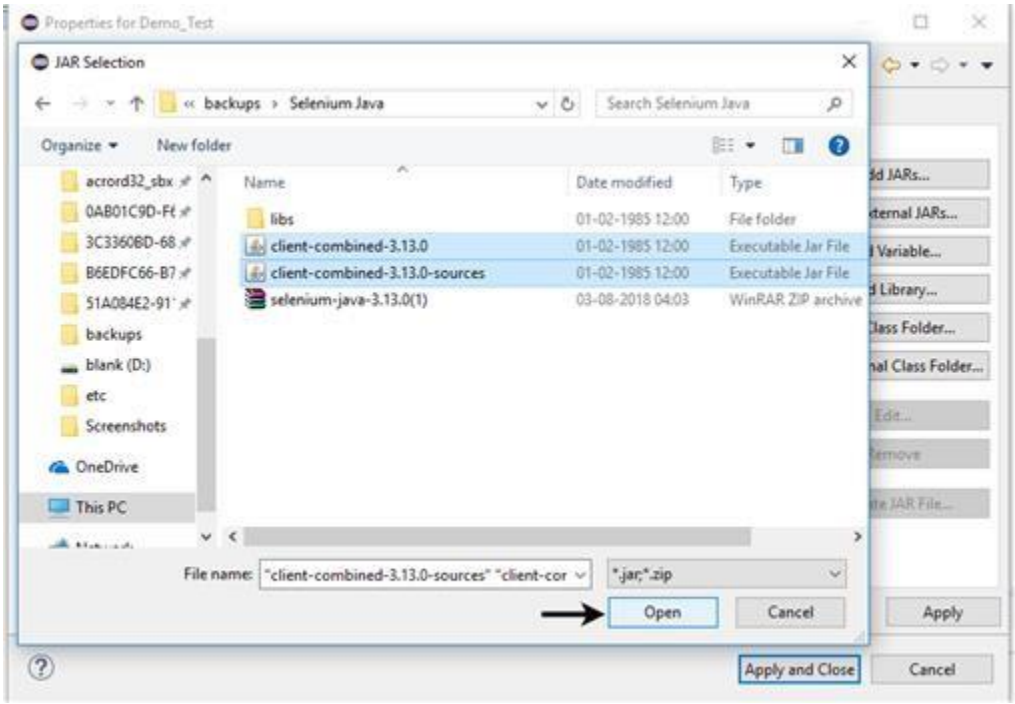
- It will launch the Properties window for our "Demo_Test" Test Suite. • Click on "Java Build Path" option from the left hand side panel.



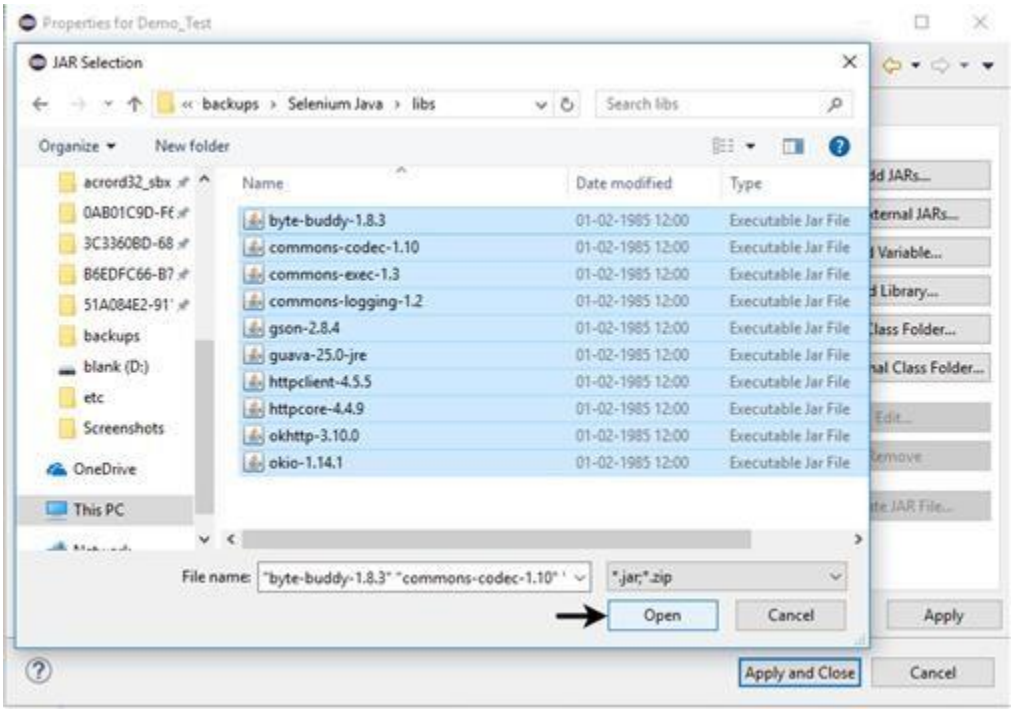
- Switch to Libraries tab and click on "Add External JARs" button.



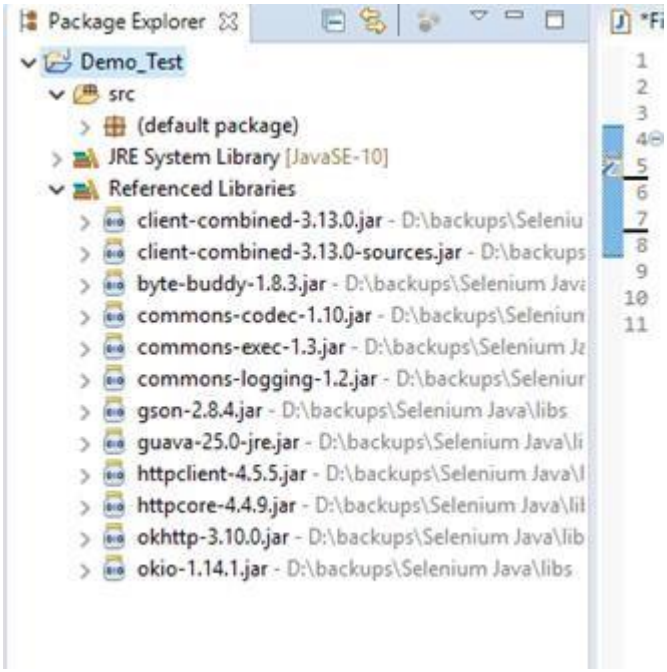
- Locate the directory where you have downloaded the Selenium jar files, select the respective jars and click on "Open" button.



- Repeat the same steps for the jars which are present under the "libs" folder. • Open "libs" folder, select all of the respective jar files and click on "Open" button.



- Once you get all the Selenium jar files in your Libraries tab, click on Apply and Close button.
- The following image shows the directory structure of our "Demo_Test" test suite after adding Selenium jars.



Hence, we have successfully configured Selenium WebDriver with Eclipse IDE. Now, we are ready to write our test scripts in Eclipse and run it in WebDriver.

Selenium WebDriver- Running test on Chrome Browser

In this section, you will learn how to run your Selenium Test Scripts on Chrome Browser.

Chrome browser implements the WebDriver protocol using an executable called **ChromeDriver.exe**. This executable start a server on your system which in turn is responsible for running your test scripts in Selenium.

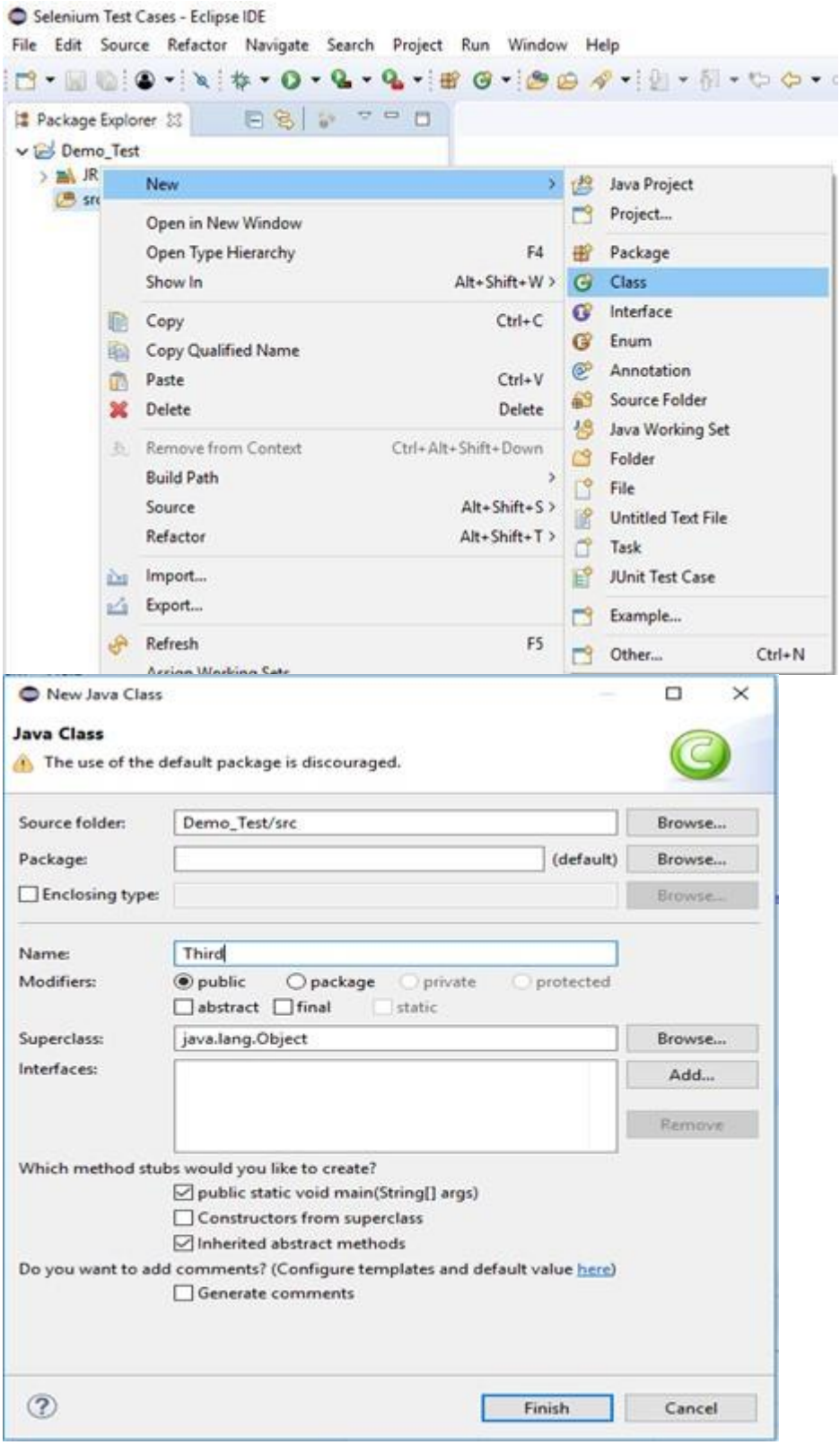
Let us consider a test case in which we will try to automate the following scenarios in Google Chrome browser.

- Launch Chrome browser.
- Maximize the browser.
- Open URL: www.javatpoint.com
- Scroll down through the web page
- Click on "Core Java" link from the Java Technology section.

We will create our third test case in the same test suite (Demo_Test).

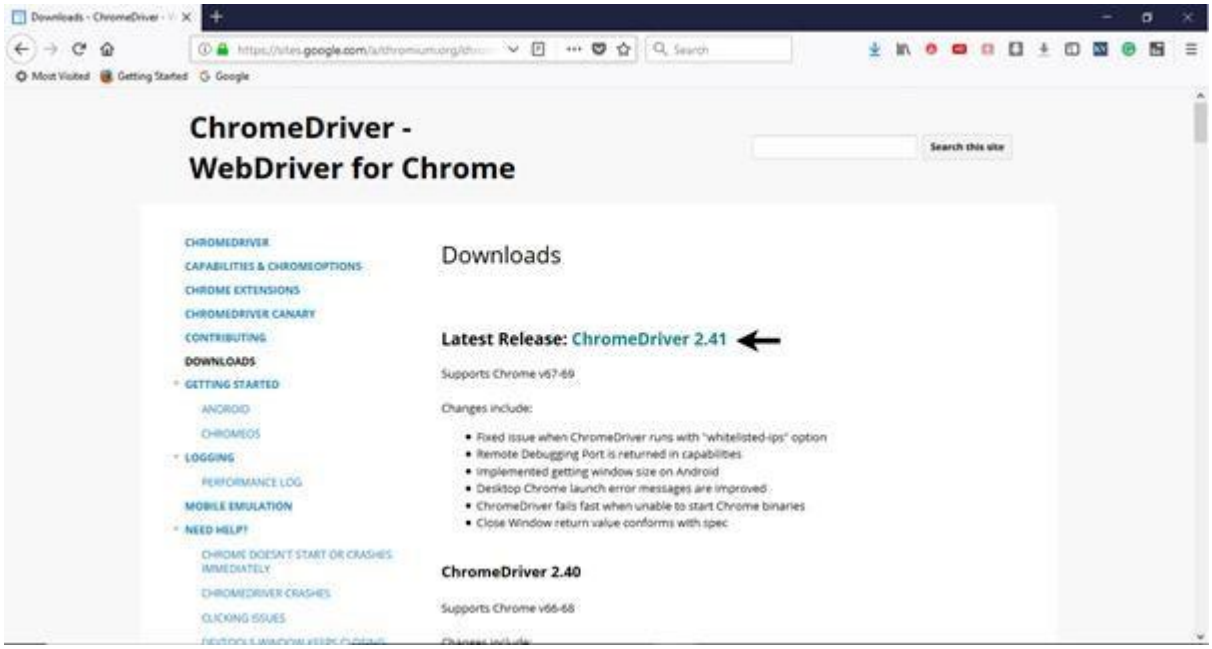
Step1. Right click on the "src" folder and create a new Class File from New > Class.

Give your Class name as "Third" and click on "Finish" button.

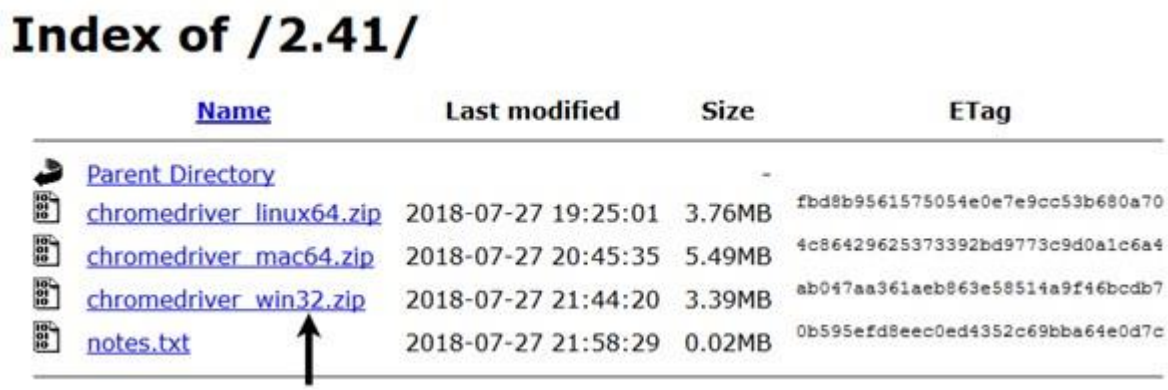


Step2. Open URL: https://sites.google.com/a/chromium.org/chromedriver/downloads in your browser.

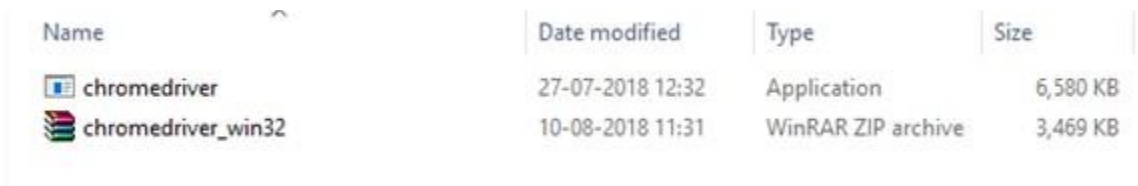
Step3. Click on the "ChromeDriver 2.41" link. It will redirect you to the directory of ChromeDriver executables files. Download as per the operating system you are working currently on.



For windows, click on the "chromedriver_win32.zip" download.



The downloaded file would be in zipped format. Unpack the contents in a convenient directory.



Step4. Set a system property "webdriver.chrome.driver" to the path of your ChromeDriver.exe file and instantiate a ChromeDriver class. Here is a sample code to do that.

```
// System Property for Chrome Driver

System.setProperty("webdriver.chrome.driver","D:\\ChromeDriver\\chromedriver.exe");
// Instantiate a ChromeDriver class.
   WebDriver driver=new ChromeDriver();
```

Step5. Now it is time to code. We have embedded comments for each block of code to explain the steps clearly.

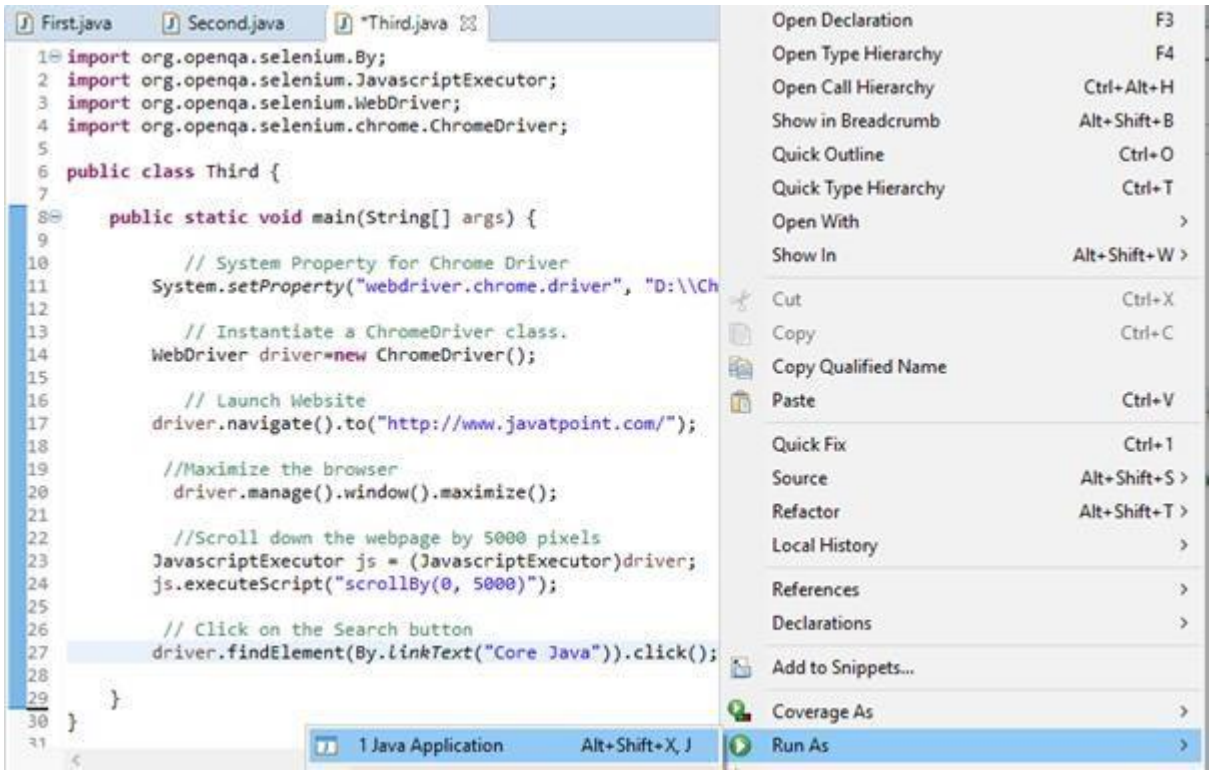
```
import org.openqa.selenium.By;
import org.openqa.selenium.JavascriptExecutor;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Third {

   public static void main(String[] args) {

        // System Property for Chrome Driver
      System.setProperty("webdriver.chrome.driver",
"D:\\ChromeDriver\\chromedriver.exe");

         // Instantiate a ChromeDriver class.
      WebDriver driver=new ChromeDriver();

        // Launch Website
      driver.navigate().to("http://www.javatpoint.com/");

       //Maximize the browser
        driver.manage().window().maximize();

       //Scroll down the webpage by 5000 pixels
      JavascriptExecutor js = (JavascriptExecutor)driver;
      js.executeScript("scrollBy(0, 5000)");

       // Click on the Search button
      driver.findElement(By.linkText("Core Java")).click();

   }

}
```

The Eclipse code window will look like this:

First.java | Second.java | *Third.java

```
import org.openqa.selenium.By;
import org.openqa.selenium.JavascriptExecutor;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Third {

    public static void main(String[] args) {

          // System Property for Chrome Driver
        System.setProperty("webdriver.chrome.driver", "D:\\ChromeDriver\\chromedriver.exe");

          // Instantiate a ChromeDriver class.
        WebDriver driver=new ChromeDriver();

          // Launch Website
        driver.navigate().to("http://www.javatpoint.com/");

         //Maximize the browser
          driver.manage().window().maximize();

          //Scroll down the webpage by 5000 pixels
        JavascriptExecutor js = (JavascriptExecutor)driver;
        js.executeScript("scrollBy(0, 5000)");

         // Click on the Search button
        driver.findElement(By.linkText("Core Java")).click();

    }
}
```

Step6. Right click on the Eclipse code and select **Run As > Java Application**.



Step6. The output of above test script would be displayed in Chrome browser.

Selenium WebDriver- Running test on Firefox Browser- Gecko (Marionette) Driver

In this section, we will learn how to run your Selenium Test Scripts on Firefox Browser.

Before proceeding with this section, let us first understand the basics of Gecko Driver.

What is Gecko Driver?

The term Gecko refers to Gecko browser engine which was developed by Mozilla Foundation as a part of Mozilla browser.

Gecko Driver serves as a link between your tests in Selenium and the Firefox browser. It acts as a proxy between W3C WebDriver-compatible clients (Eclipse, Netbeans, etc.) to interact with Gecko-based browser (Mozilla Firefox).

Marionette (the next generation of FirefoxDriver) is turned on by default from Selenium 3. Selenium uses W3C Webdriver protocol to send requests to GeckoDriver, which translates them into a protocol named Marionette. Even if you are working with older versions of Firefox browser, Selenium 3 expects you to set path to the driver executable by the **webdriver.gecko.driver**.

Note: Selenium 3 has upgraded itself to now launch Firefox driver using Marionette driver instead of the default initialisation supported earlier.

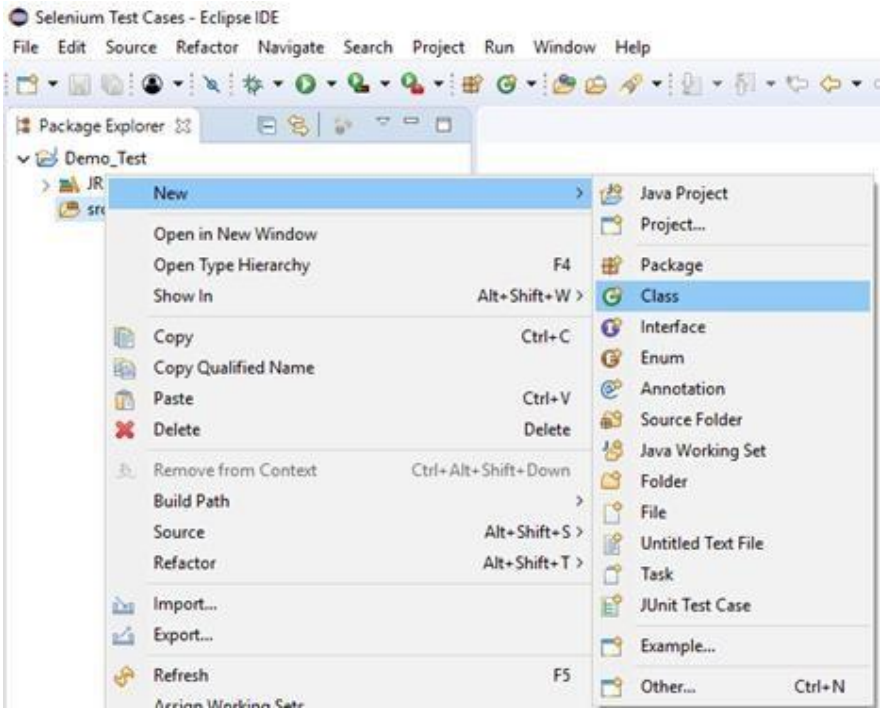
Let us consider a test case in which we will try to automate the following scenarios in Firefox browser.

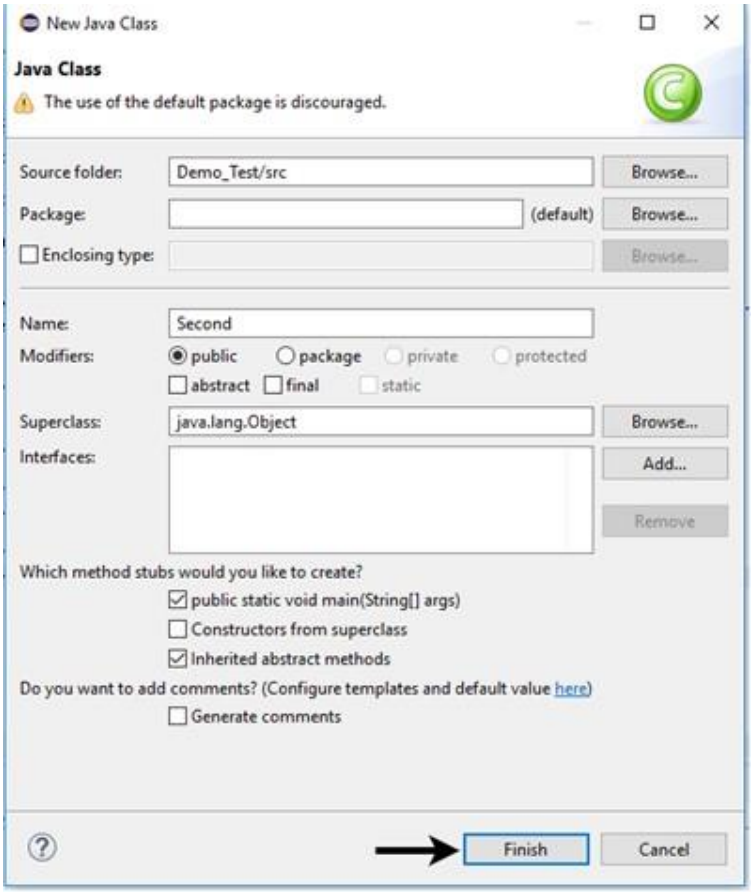
- Launch Firefox browser.
- Open URL: www.javatpoint.com
- Click on the Custom Search text box
- Type the value "Java"
- Click on the Search button.

We will create our second test case in the same test suite (Demo_Test).

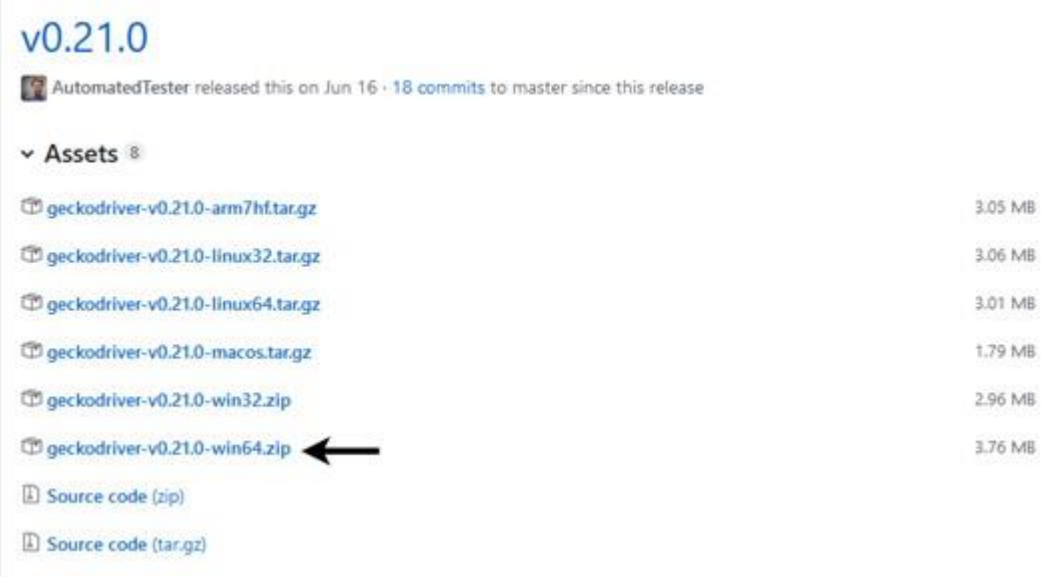
Step1. Right click on the "src" folder and create a new Class File from New > Class.

Give your Class name as "Second" and click on "Finish" button.

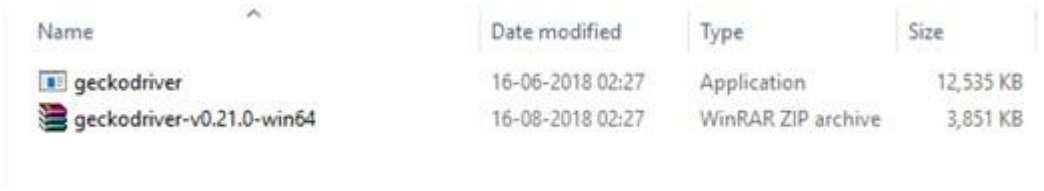

Selenium Test Cases - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Demo_Test
New
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Java Project
Project...
Package
Class
Interface
Enum
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
Task
JUnit Test Case
Example...
Other... Ctrl+N


New Java Class
Java Class
The use of the default package is discouraged.
Source folder: Demo_Test/src
Browse...
Package:
(default)
Enclosing type:
Name: Second
Modifiers: public package private protected
abstract final static
Superclass: java.lang.Object
Interfaces:
Add...
Remove
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass
Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
Generate comments
Finish
Cancel

Step2. Open URL: https://github.com/mozilla/geckodriver/releases in your browser and click on the appropriate version for GeckoDriver download based on the operating system you are currently working on. Here, we are downloading the 64bit version of GeckoDriver for windows.



The downloaded file would be in zipped format. Unpack the contents in a convenient directory.



Before writing the test script, let us first understand how we can initialize GeckoDriver in Selenium. There are three ways to initialize GeckoDriver:

1. Using Desired Capabilities

First, we have to set the system property for Gecko Driver.

```
System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );
```

Below is the code to set gecko driver using DesiredCapabilities class.

```
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
```

```
capabilities.setCapability("marionette",true);
```

Here is the complete code:

System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe");
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
capabilities.setCapability("marionette",true); WebDriver driver= new FirefoxDriver(capabilities);

2. Using marionette property:

Gecko Driver can also be initialized using marionette property.
System.setProperty("webdriver.firefox.marionette","D:\\GeckoDriver\\geckodriver.exe");

The code for Desired Capabilities is not required for this method.

3. Using Firefox Options:

Firefox 47 or later versions have marionette driver as a legacy system. Thus, marionette driver can be called using Firefox Options as shown below. FirefoxOptions options = new FirefoxOptions(); options.setLegacy(true);

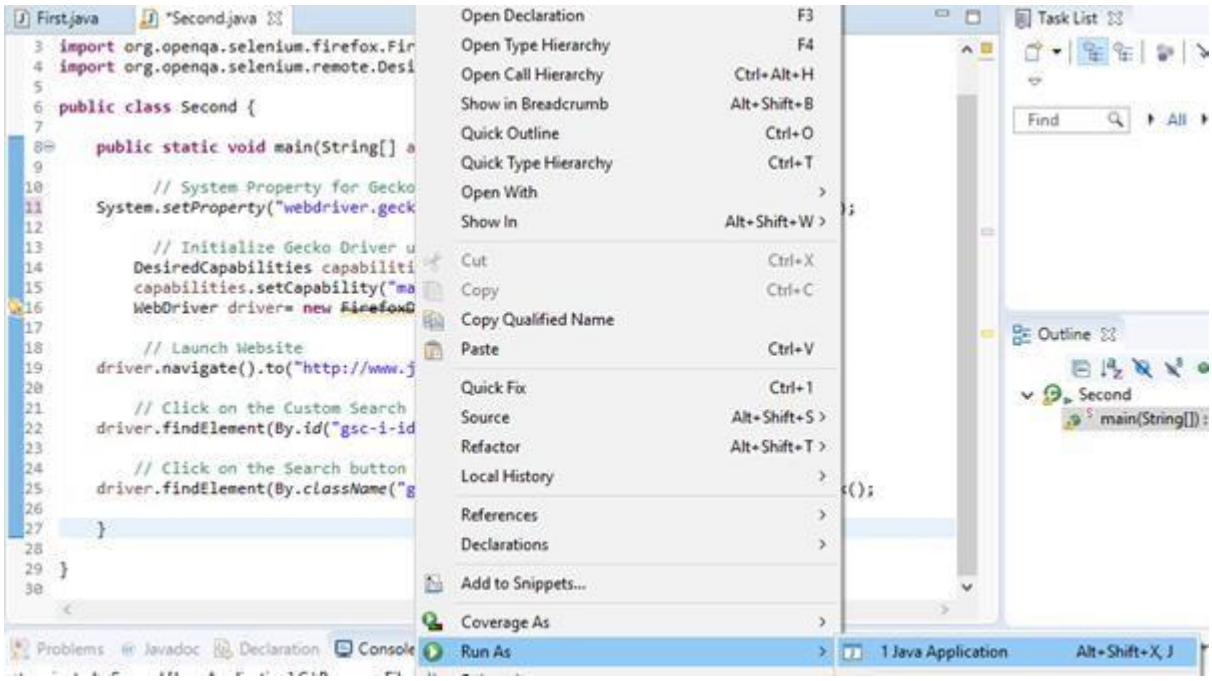
Step3. Now it is time to code. We have embedded comments for each block of code to explain the steps clearly.

```
import org.openqa.selenium.By; import
org.openqa.selenium.WebDriver; import
org.openqa.selenium.firefox.FirefoxDriver; import
org.openqa.selenium.remote.DesiredCapabilities;

public class Second {

  public static void main(String[] args) {

      // System Property for Gecko Driver
System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );

      // Initialize Gecko Driver using Desired Capabilities Class
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
  capabilities.setCapability("marionette",true);
WebDriver driver= new FirefoxDriver(capabilities);

     // Launch Website
  driver.navigate().to("http://www.javatpoint.com/");

    // Click on the Custom Search text box and send value
driver.findElement(By.id("gsc-i-id1")).sendKeys("Java");
```

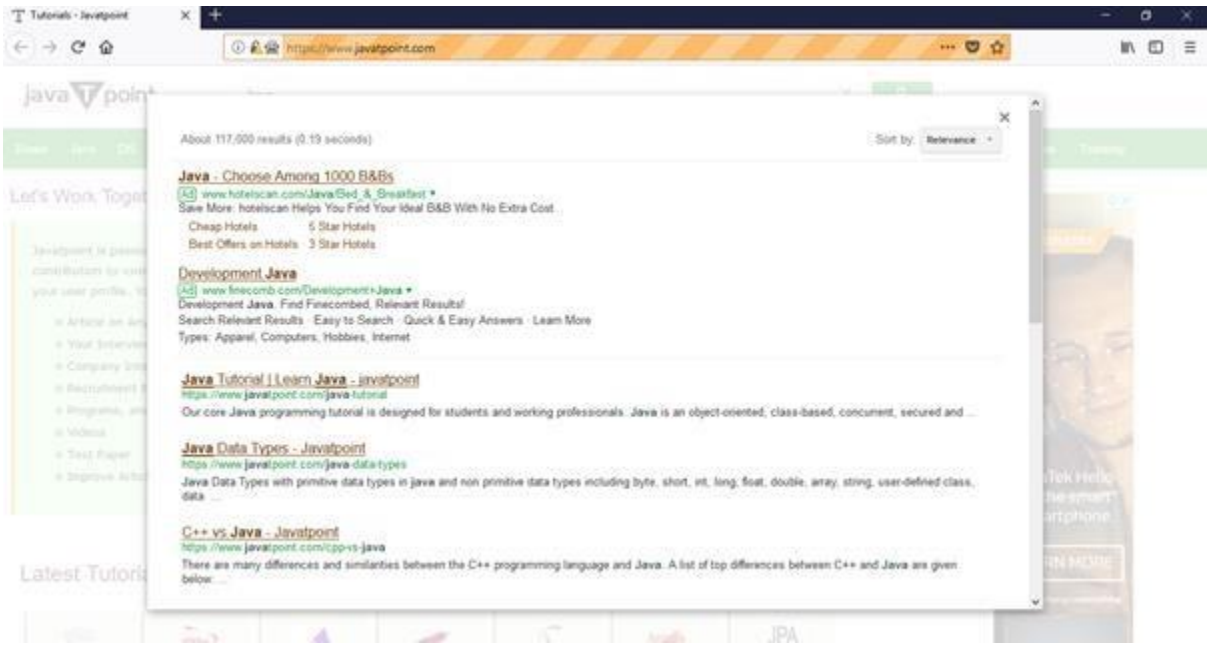
```
      // Click on the Search button
driver.findElement(By.className("gsc-search-button gsc-search-buttonv2")).click();
      }

}
```

The Eclipse code window will look like this:

Step4. Right click on the Eclipse code and select **Run As > Java Application**.



Step5. The output of above test script would be displayed in Firefox browser.



Selenium WebDriver- Drag and Drop

In this section, you will learn how to perform complex operation like Drag and Drop in Selenium WebDriver.

Before proceeding with this section, let us first understand some of the concepts regarding Drag and Drop operation.

Actions in Selenium WebDriver

For performing complex user interaction like drag and drop, we have an Actions class in Selenium WebDriver. Using the Actions class, we first build a sequence of composite events and then perform it using Action (an interface which represents a single user-interaction). The different methods of Actions class we will be using here are-

- **clickAndHold(WebElement element)** - Clicks a web element at the middle(without releasing).
- **moveToElement(WebElement element)** - Moves the mouse pointer to the middle of the web element without clicking.
- **release(WebElement element)** - Releases the left click (which is in pressed state).
- **build()** - Generates a composite action Let us consider a test case in which we will

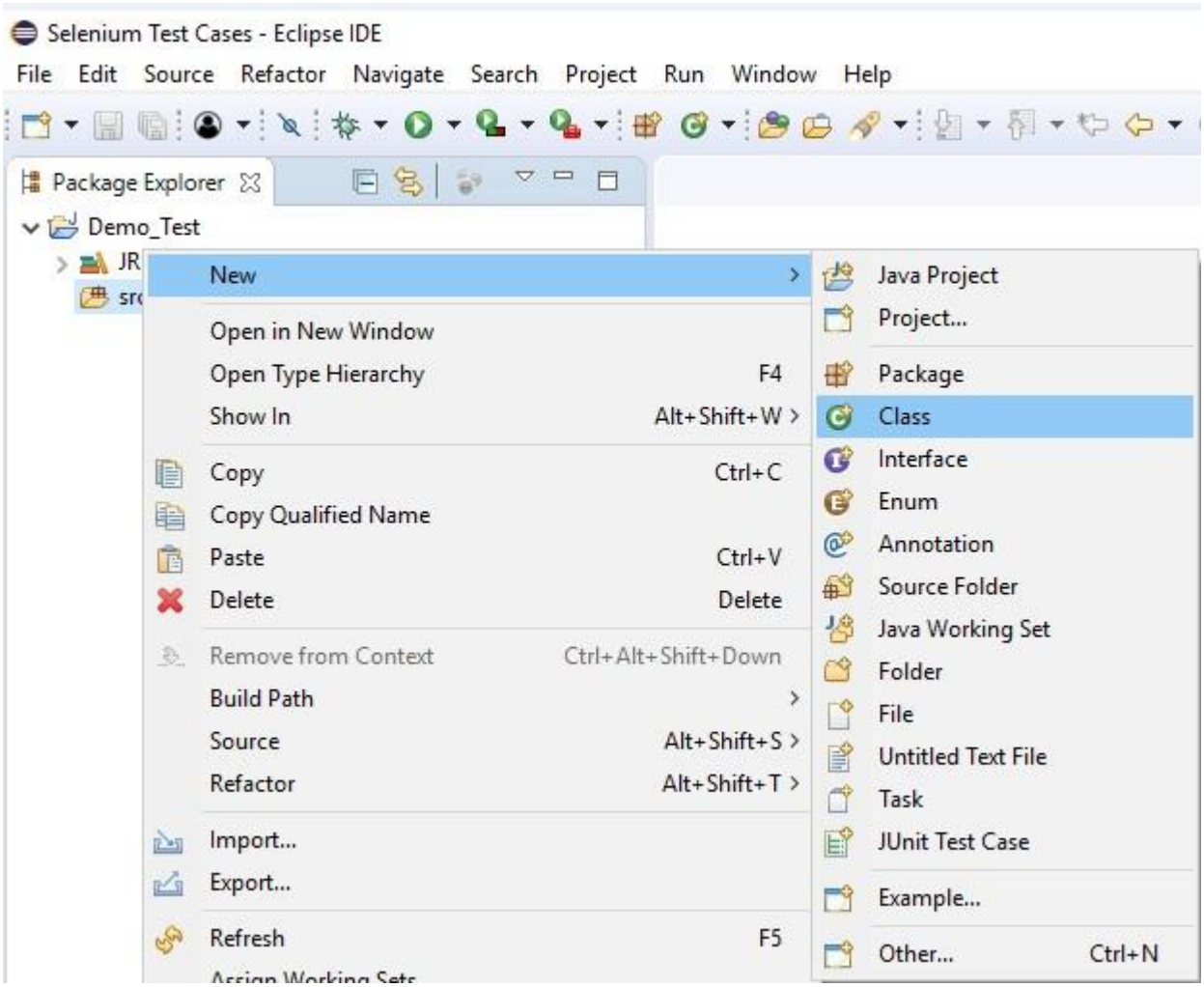
  automate the following scenarios:

- Invoke Firefox Browser
- Open URL: https://www.testandquiz.com/selenium/testing.html
- Drag and Drop the JavaTpoint icon on the textbox

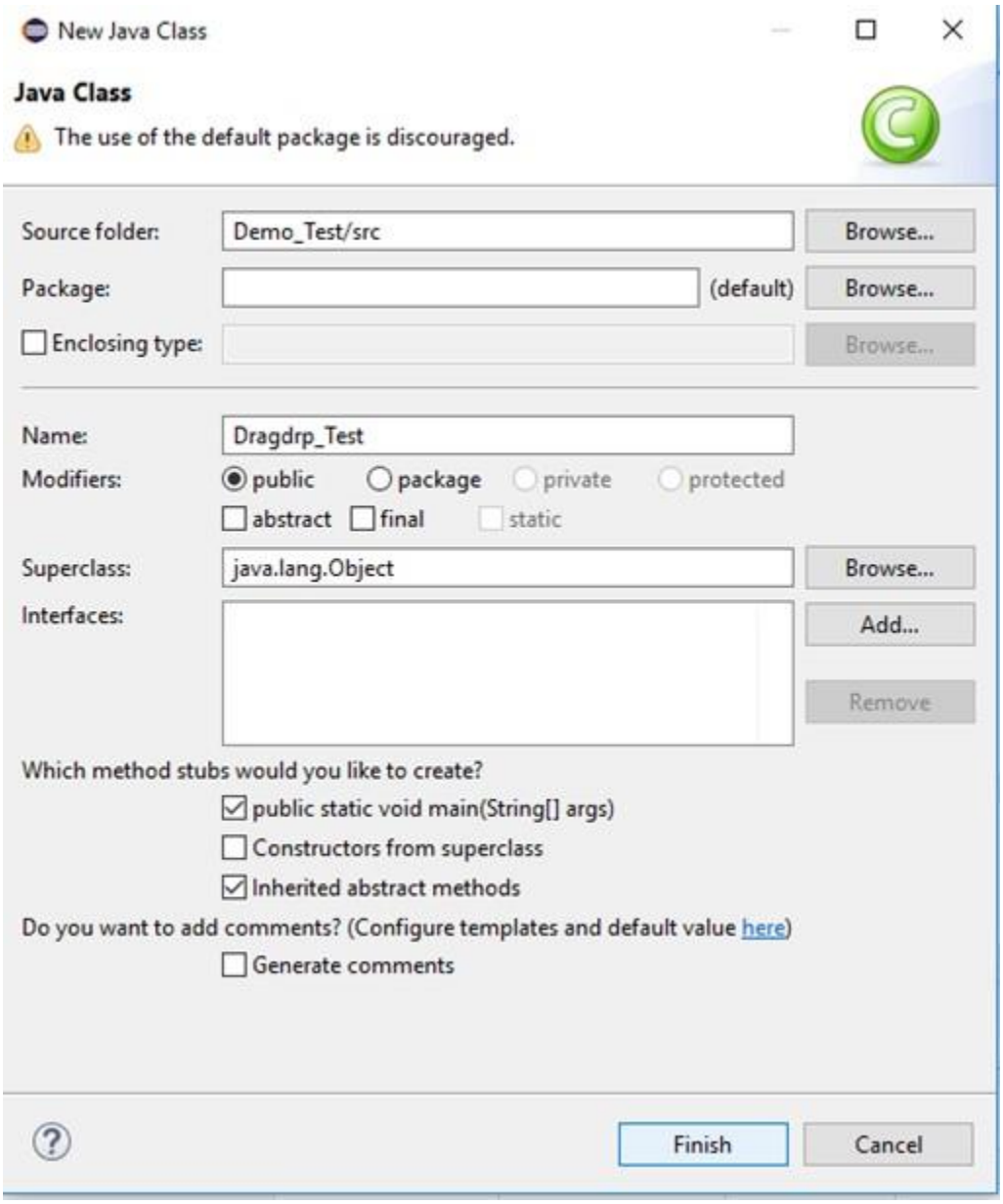
We will create our test case step by step in order to give you a complete understanding of how to handle drag and drop in WebDriver.

Step1. Launch Eclipse IDE and open the existing test suite "Demo_Test" which we have created in earlier sessions of this tutorial.

Step2. Right click on the "src" folder and create a new Class File from **New >Class**.



Give your Class name as "Dragdrp_Test" and click on "Finish" button.



Step3. Let's get to the coding ground.

- To invoke Firefox browser, we need to download Gecko driver and set the system property for Gecko driver. We have already discussed this in earlier sessions of this tutorial. You can refer "Running test on Firefox Browser" to learn how to download and set System property for Firefox driver.

Here is the sample code to set system property for Gecko driver:

```
// System Property for Gecko Driver
ystem.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );
```

- After that we have to initialize Gecko Driver using Desired Capabilities Class.

Here is the sample code to initialize gecko driver using DesiredCapabilities class.

```
// Initialize Gecko Driver using Desired Capabilities Class
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
capabilities.setCapability("marionette",true);
WebDriver driver= new FirefoxDriver(capabilities);
```

Combining both of the above code blocks, we will get the code snippet to launch Firefox browser.

```
 // System Property for Gecko Driver
System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );

 // Initialize Gecko Driver using Desired Capabilities Class   DesiredCapabilities
capabilities = DesiredCapabilities.firefox();
capabilities.setCapability("marionette",true);   WebDriver
driver= new FirefoxDriver(capabilities);
```

After that we need to write the code which will automate our second test scenario (navigate to the desired URL)

Here is the sample code to navigate to the desired URL:

```
// Launch Website
driver.navigate().to("https://www.testandquiz.com/selenium/testing.html");   The
```

complete code till now will look something like this:

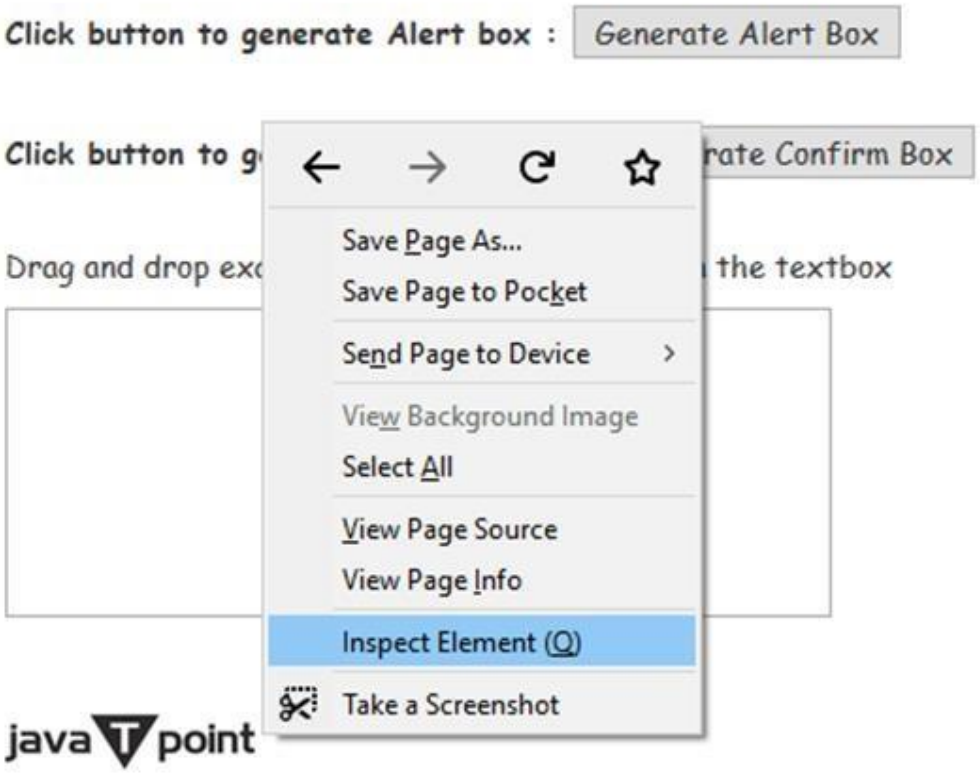
```
import org.openqa.selenium.WebDriver;   import
org.openqa.selenium.firefox.FirefoxDriver;   import
org.openqa.selenium.remote.DesiredCapabilities;

public class Dragdrp_Test {

   public static void main(String[] args) {

   // System Property for Gecko Driver
      System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );

   // Initialize Gecko Driver using Desired Capabilities Class
         DesiredCapabilities capabilities = DesiredCapabilities.firefox();
         capabilities.setCapability("marionette",true);
WebDriver driver= new FirefoxDriver(capabilities);
    // Launch Website
driver.navigate().to("https://www.testandquiz.com/selenium/testing.html");

   }
```

}

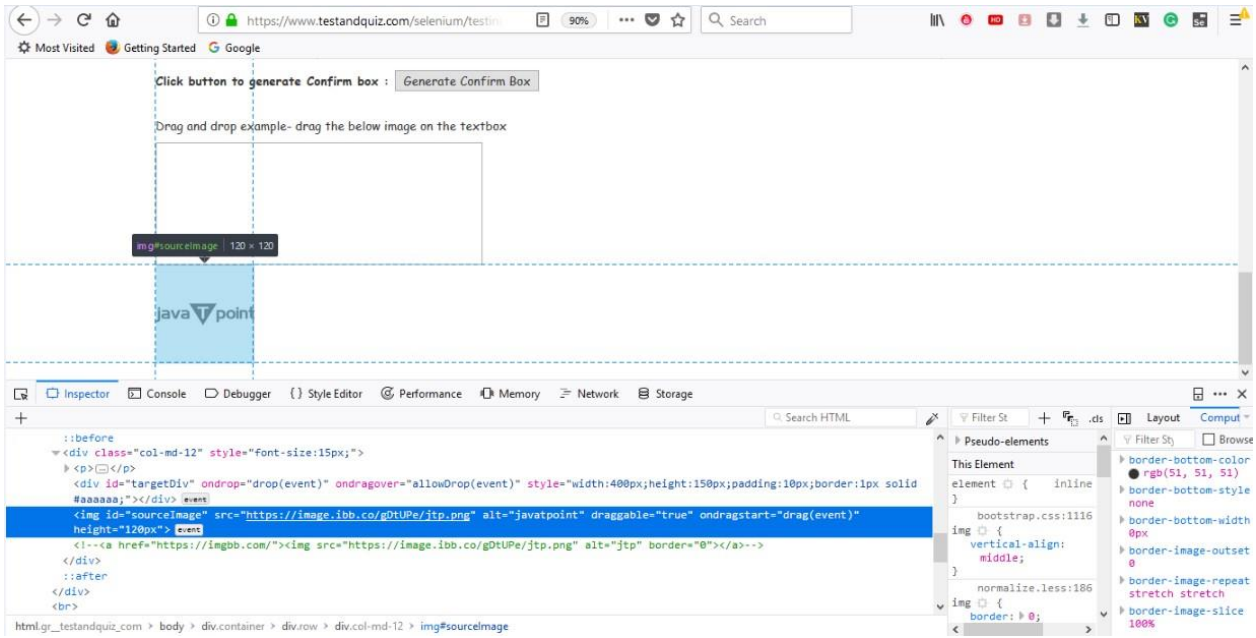
Step4. Now we will try to locate the JavaTpoint icon and text box in order to perform drag and drop operation. As we know that locating an element involves inspection of its HTML codes.

Follow the steps given below to locate the drop-down menu on the sample web page.

- Open URL: https://www.testandquiz.com/selenium/testing.html • Right click on JavaTpoint icon and select Inspect Element.

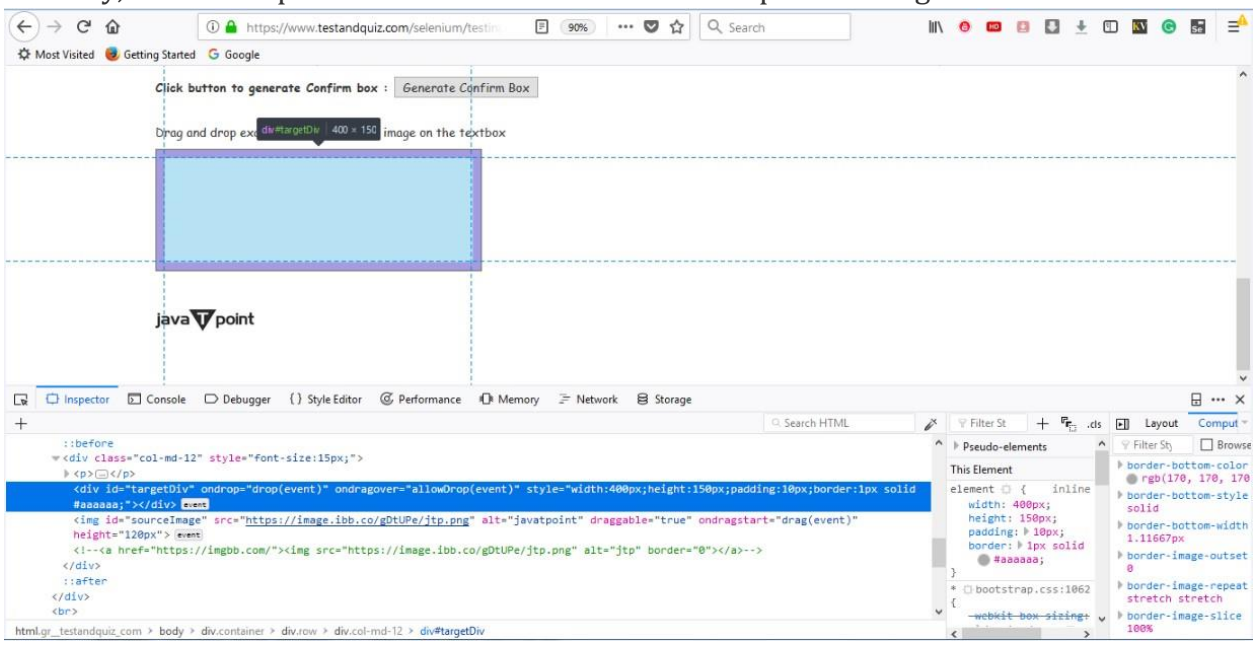


- It will launch a window containing all the specific codes involved in the development of the JavaTpoint logo.

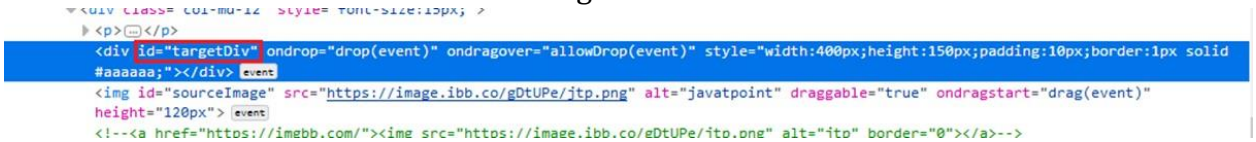


- Take a note of its id attribute i.e. sourceImage

Similarly, we will inspect the text box where we have to place the logo.



- Take a note of its id attribute i.e. targetDiv



Step5. To automate our third and fourth test scenario, we need to write the code which will perform the drag and drop operation on the JavaTpoint logo. Given below is the code snippet to perform drag and drop operation.

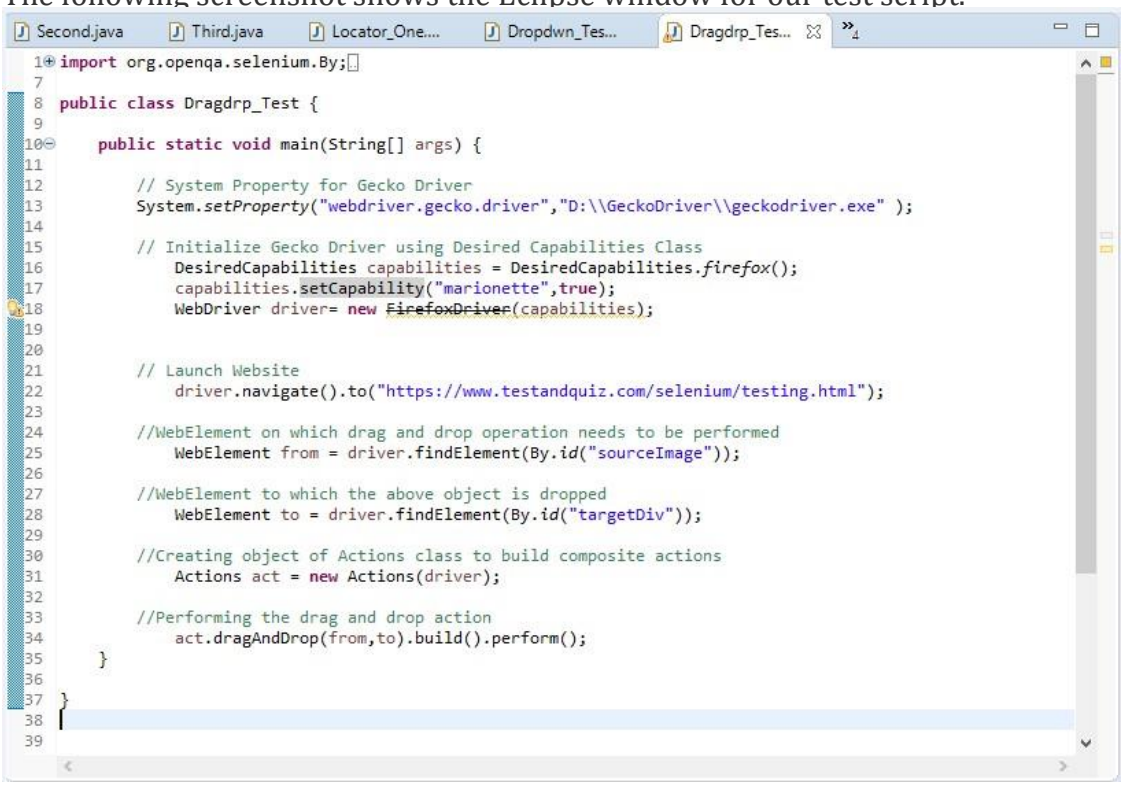
```
//WebElement on which drag and drop operation needs to be performed
WebElement from = driver.findElement(By.id("sourceImage"));

//WebElement to which the above object is dropped
WebElement to = driver.findElement(By.id("targetDiv");

//Creating object of Actions class to build composite actions
Actions act = new Actions(driver);

//Performing the drag and drop action
act.dragAndDrop(from,to).build().perform();
```

Thus, our final test script will look something like this:

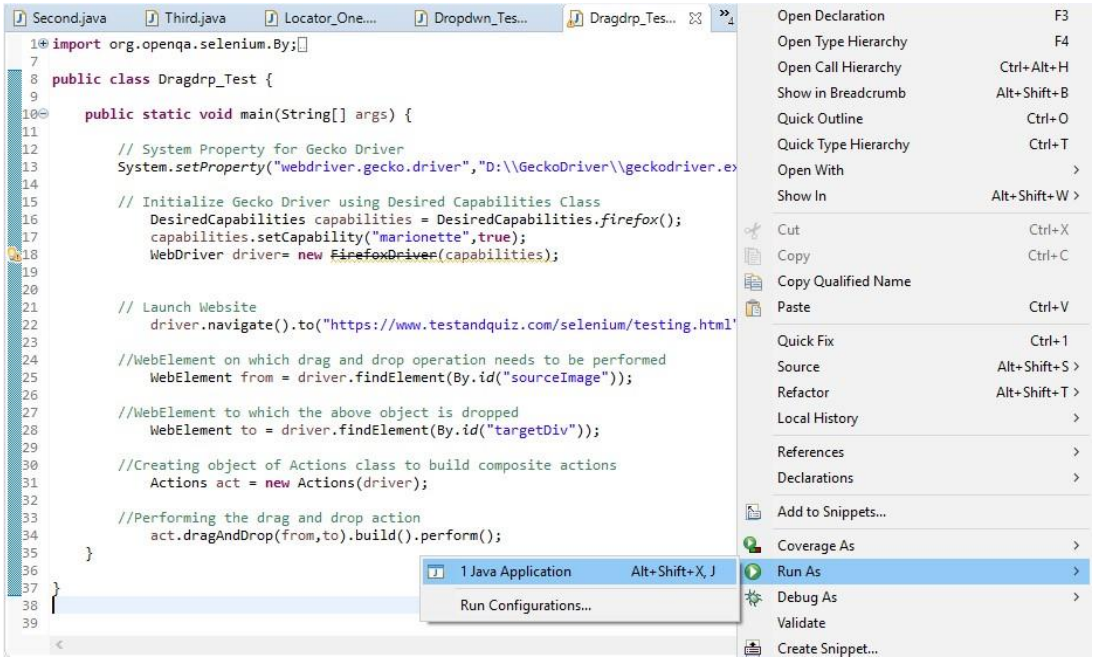
```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;
import org.openqa.selenium.remote.DesiredCapabilities;

public class Dragdrp_Test {

   public static void main(String[] args) {

      // System Property for Gecko Driver
      System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );

      // Initialize Gecko Driver using Desired Capabilities Class
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
         capabilities.setCapability("marionette",true);
WebDriver driver= new FirefoxDriver(capabilities);
      // Launch Website
         driver.navigate().to("https://www.testandquiz.com/selenium/testing.html");

      //WebElement on which drag and drop operation needs to be performed
         WebElement from = driver.findElement(By.id("sourceImage"));

      //WebElement to which the above object is dropped
```

```
        WebElement to = driver.findElement(By.id("targetDiv"));

      //Creating object of Actions class to build composite actions
        Actions act = new Actions(driver);

      //Performing the drag and drop action
        act.dragAndDrop(from,to).build().perform();
   }
}
```

The following screenshot shows the Eclipse window for our test script.

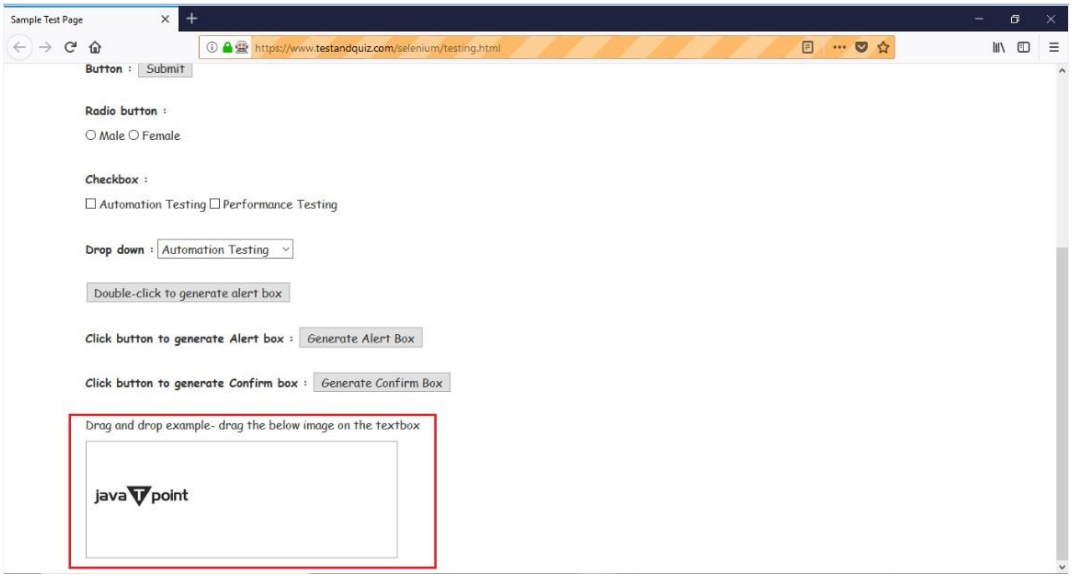


Step6. Right click on the Eclipse code and select **Run As > Java Application**.



Upon execution, the above test script will launch the Firefox browser and automate all the test scenarios.

You can see that the JavaTpoint logo will automatically get dropped into the text box.



Selenium WebDriver- Handling Alerts

In this section, you will learn how to handle alerts in Selenium WebDriver.

Selenium WebDriver provides three methods to accept and reject the Alert depending on the Alert types.

1. void dismiss()

This method is used to click on the 'Cancel' button of the alert.

Syntax: driver.switchTo().alert().dismiss();

2. void accept()

This method is used to click on the 'Ok' button of the alert.

Syntax: driver.switchTo().alert().accept();

3. String getText()

This method is used to capture the alert message.

Syntax: driver.switchTo().alert().getText();

4. void sendKeys(String stringToSend)

This method is used to send some data to the alert box.

Syntax: driver.switchTo().alert().sendKeys("Text");

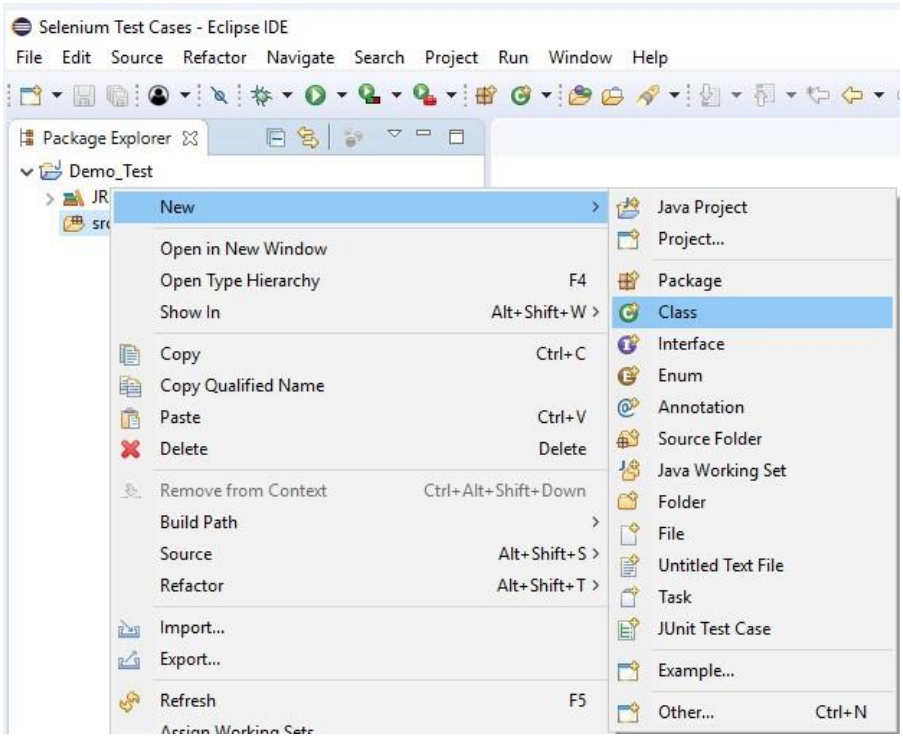
Let us consider a test case in which we will automate the following scenarios:

- Invoke Firefox Browser
- Open URL: https://www.testandquiz.com/selenium/testing.html
- Click on the "Generate Alert box" button
- Click on the "Generate Confirm box" button
- Close the browser

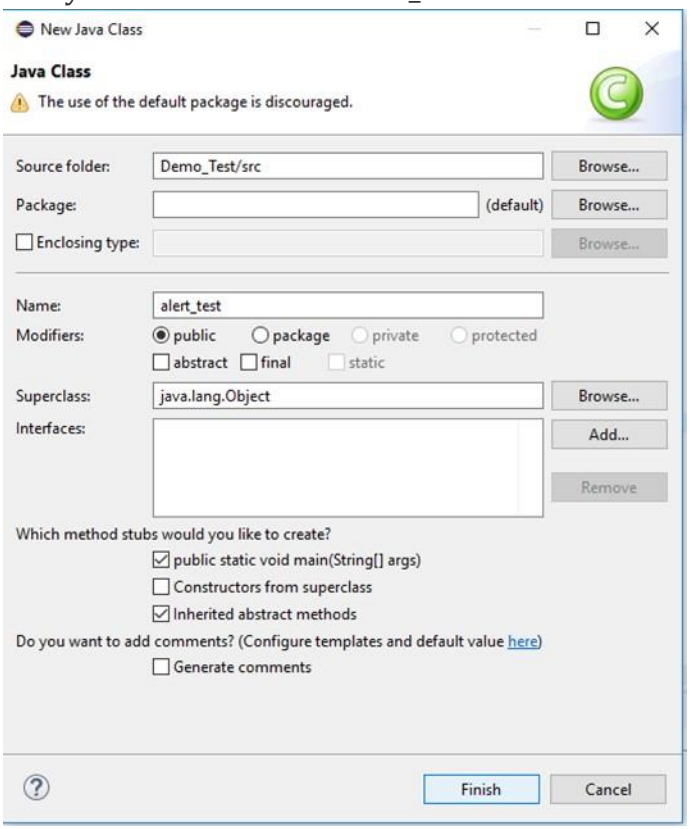
We will create our test case step by step in order to give you a complete understanding of how to handle alerts in WebDriver.

Step1. Launch Eclipse IDE and open the existing test suite "Demo_Test" which we have created in earlier sessions of this tutorial.

Step2. Right click on the "src" folder and create a new Class File from **New >Class**.



Give your Class name as "alert_test" and click on "Finish" button.



Step3. Let's get to the coding ground.

- To invoke Firefox browser, we need to download Gecko driver and set the system property for Gecko driver. We have already discussed this in earlier sessions of this tutorial. You can refer "Running test on Firefox Browser" to learn how to download and set System property for Firefox driver. Here is the sample code to set system property for Gecko driver:

```
// System Property for Gecko Driver
stem.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );
```

- After that we have to initialize Gecko Driver using Desired Capabilities Class. Here is the sample code to initialize gecko driver using DesiredCapabilities class.

```
// Initialize Gecko Driver using Desired Capabilities Class
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
capabilities.setCapability("marionette",true);   ebDriver
driver= new FirefoxDriver(capabilities);
```

Combining both of the above code blocks, we will get the code snippet to launch Firefox browser.

```
 // System Property for Gecko Driver
System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );

 // Initialize Gecko Driver using Desired Capabilities Class
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
capabilities.setCapability("marionette",true);
WebDriver driver= new FirefoxDriver(capabilities);
```

- After that we need to write the code which will automate our second test scenario (navigate to the desired URL)

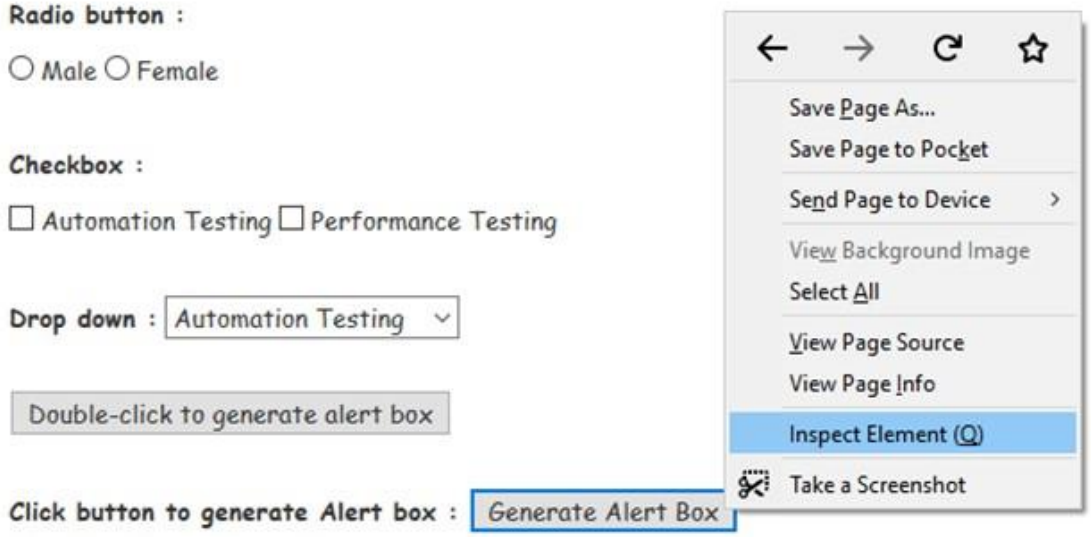
Here is the sample code to navigate to the desired URL:

```
// Launch Website
driver.navigate().to("https://www.testandquiz.com/selenium/testing.html");
The complete code till now will look something like this: import
org.openqa.selenium.WebDriver;  import
org.openqa.selenium.firefox.FirefoxDriver;  import
org.openqa.selenium.remote.DesiredCapabilities;

public class alert_test {

  public static void main(String[] args) {

  // System Property for Gecko Driver
     System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );
  // Initialize Gecko Driver using Desired Capabilities Class
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
capabilities.setCapability("marionette",true);
        WebDriver driver= new FirefoxDriver(capabilities);
   // Launch Website
   driver.navigate().to("https://www.testandquiz.com/selenium/testing.html");
  }
}
```

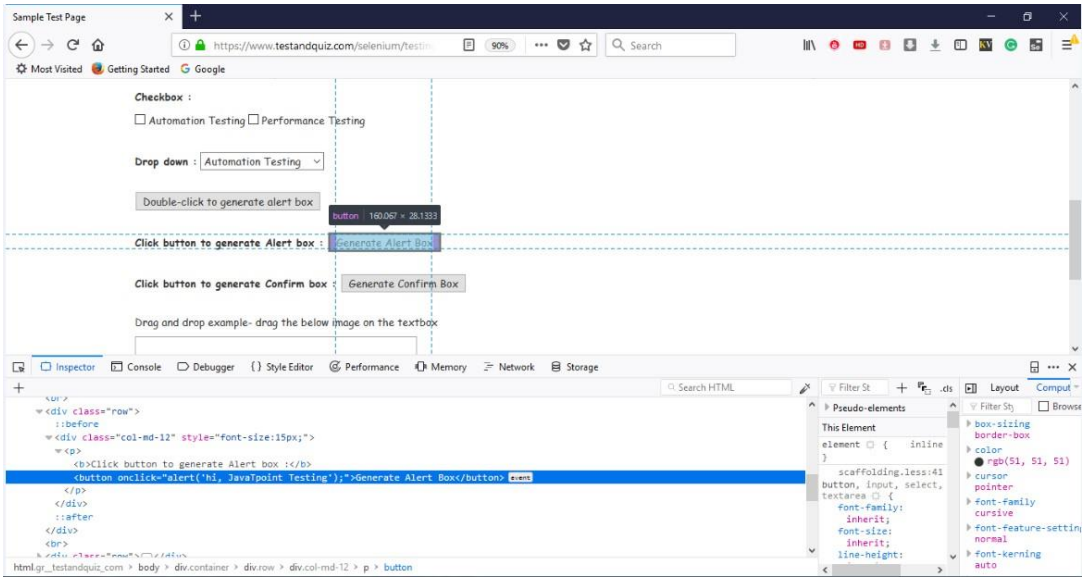
Step4. Now, we will try to locate the "Generate Alert Box" and "Generate Confirm Box" in order to perform alert handling operation. As we know that locating an element involves inspection of its HTML codes.

Follow the steps given below to locate the drop-down menu on the sample web page.

- Open URL: https://www.testandquiz.com/selenium/testing.html
- Right click on the "Generate Alert Box" button and select Inspect Element.

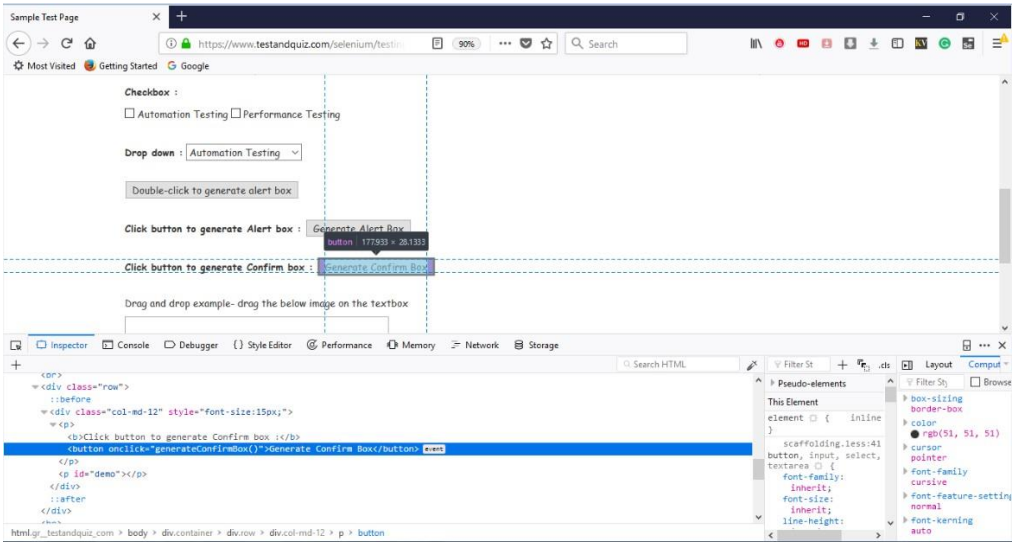


- It will launch a window containing all the specific codes involved in the development of the "Generate Alert Box" button.



- Take a note of its link text i.e. "Generate Alert Box"

Similarly, we will inspect the "Generate Confirm box" button.



- Take a note of its link text i.e. "Generate Confirm Box"

Step5.

To automate our third and fourth test scenario, we need to write the code which click and accept the **Generate Alert box** as well as click and accept the Generate Confirm box.

Given below is the code snippet to automate our third and fourth test scenario.

```
//Handling alert boxes
   //Click on generate alert button
    driver.findElement(By.linkText("Generate Alert Box")).click();

   //Using Alert class to first switch to or focus to the alert box
    Alert alert = driver.switchTo().alert();
   //Using accept() method to accep the alert box
alert.accept();
   //Handling confirm box
   //Click on Generate Confirm Box
    driver.findElement(By.linkText("Generate Confirm Box")).click();
    Alert confirmBox = driver.switchTo().alert();
   //Using dismiss() command to dismiss the confirm box
//Similarly accept can be used to accept the confirm box
confirmBox.dismiss();
```

Thus, our final test script will look something like this:

```
import org.openqa.selenium.By;   import                                    );
org.openqa.selenium.WebDriver;   import
org.openqa.selenium.firefox.FirefoxDriver;   import
org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.Alert;
public class alert_test {

   public static void main(String[] args) {

      // System Property for Gecko Driver

System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe"
// Initialize Gecko Driver using Desired Capabilities Class
         DesiredCapabilities capabilities = DesiredCapabilities.firefox();
         capabilities.setCapability("marionette",true);
WebDriver driver= new FirefoxDriver(capabilities);


      // Launch Website
      driver.navigate().to("https://www.testandquiz.com/selenium/testing.html");

      //Handling alert boxes
      //Click on generate alert button
      driver.findElement(By.linkText("Generate Alert Box")).click();

      //Using Alert class to first switch to or focus to the alert box
      Alert alert = (Alert) driver.switchTo().alert();
//Using accept() method to accept the alert box
alert.accept();
      //Handling confirm box
      //Click on Generate Confirm Box
      driver.findElement(By.linkText("Generate Confirm Box")).click();
      Alert confirmBox = (Alert) driver.switchTo().alert();
      //Using dismiss() command to dismiss the confirm box
      //Similarly accept can be used to accept the confirm box
      ((Alert) confirmBox).dismiss();
   }
}
```

Upon execution, the above test script will launch the Firefox browser and automate all the test scenarios.

Selenium WebDriver - Browser Commands

The very basic browser operations of WebDriver include **opening a browser**; perform few tasks and then **closing the browser**.

Given are some of the most commonly used Browser commands for Selenium WebDriver.

1. Get Command

Method: get(String arg0) : **void**

In WebDriver, this method loads a new web page in the existing browser window. It accepts *String* as parameter and returns *void*.

The respective command to load a new web page can be written as:

```
driver.get(URL);

// Or can be written as

String URL = "URL";
driver.get(URL);
```

Example: For instance, the command to load the official website of javaTpoint can be written as:

```
driver.get("www.javatpoint.com")
```

2. Get Title Command

Method: getTitle(): String

In WebDriver, this method fetches the title of the current web page. It accepts no parameter and returns a String.
The respective command to fetch the title of the current page can be written as:

```
driver.getTitle();
  // Or can be written as
String Title = driver.getTitle();
```

3. Get Current URL Command

Method: getCurrentUrl(): String

In WebDriver, this method fetches the string representing the Current URL of the current web page. It accepts nothing as parameter and returns a String value. The respective

command to fetch the string representing the current URL can be written as:
driver.getCurrentUrl(); //Or can be written as
String CurrentUrl = driver.getCurrentUrl();

4. Get Page Source Command

Method: getPageSource(): String

In WebDriver, this method returns the source code of the current web page loaded on the current browser. It accepts nothing as parameter and returns a *String* value. The respective command to get the source code of the current web page can be written as:
driver.getPageSource(); //Or can be written as
String PageSource = driver.getPageSource();

5. Close Command

Method: close(): **void**

This method terminates the current browser window operating by WebDriver at the current time. If the current window is the only window operating by WebDriver, it terminates the browser as well. This method accepts nothing as parameter and returns *void*. The respective command to terminate the browser window can be written as:
driver.close();

6. Quit Command

Method: quit(): **void**
This method terminates all windows operating by WebDriver. It terminates all tabs as well as the browser itself. It accepts nothing as parameter and returns void.

The respective command to terminate all windows can be written as: driver.quit(); Let us consider a sample test script in which will cover most of the Browser Commands provided by WebDriver.

In this sample test, we will automate the following test scenarios:

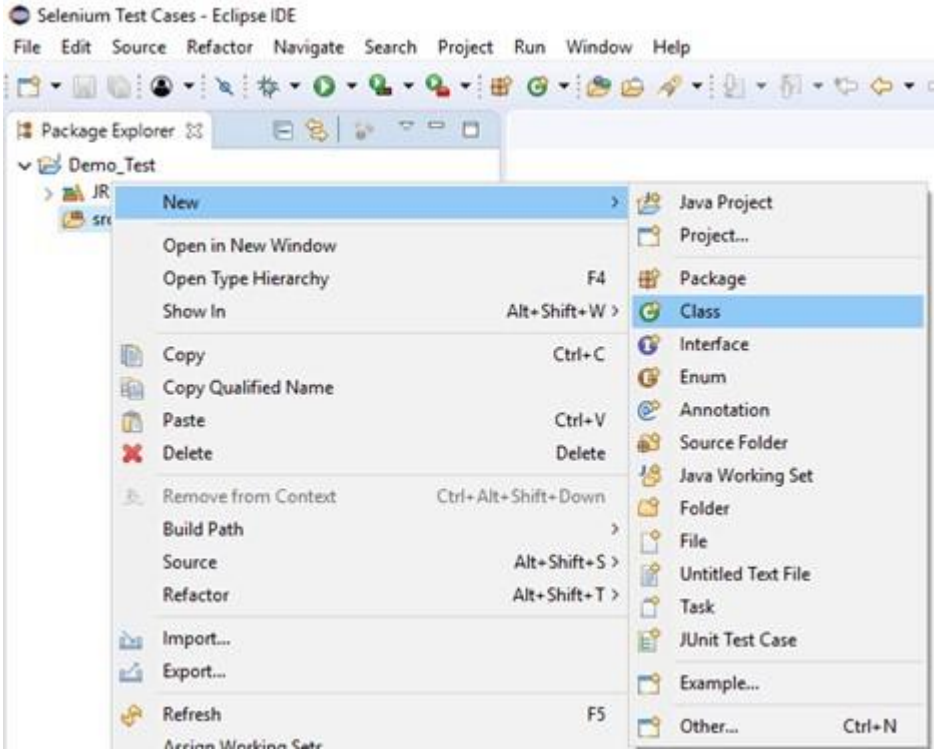
- Invoke Chrome Browser
- Open URL: https://www.google.co.in/
- Get Page Title name and Title length

- Print Page Title and Title length on the Eclipse Console
- Get page URL and verify whether it is the desired page or not • Get page Source and Page Source length
- Print page Length on Eclipse Console.
- Close the Browser

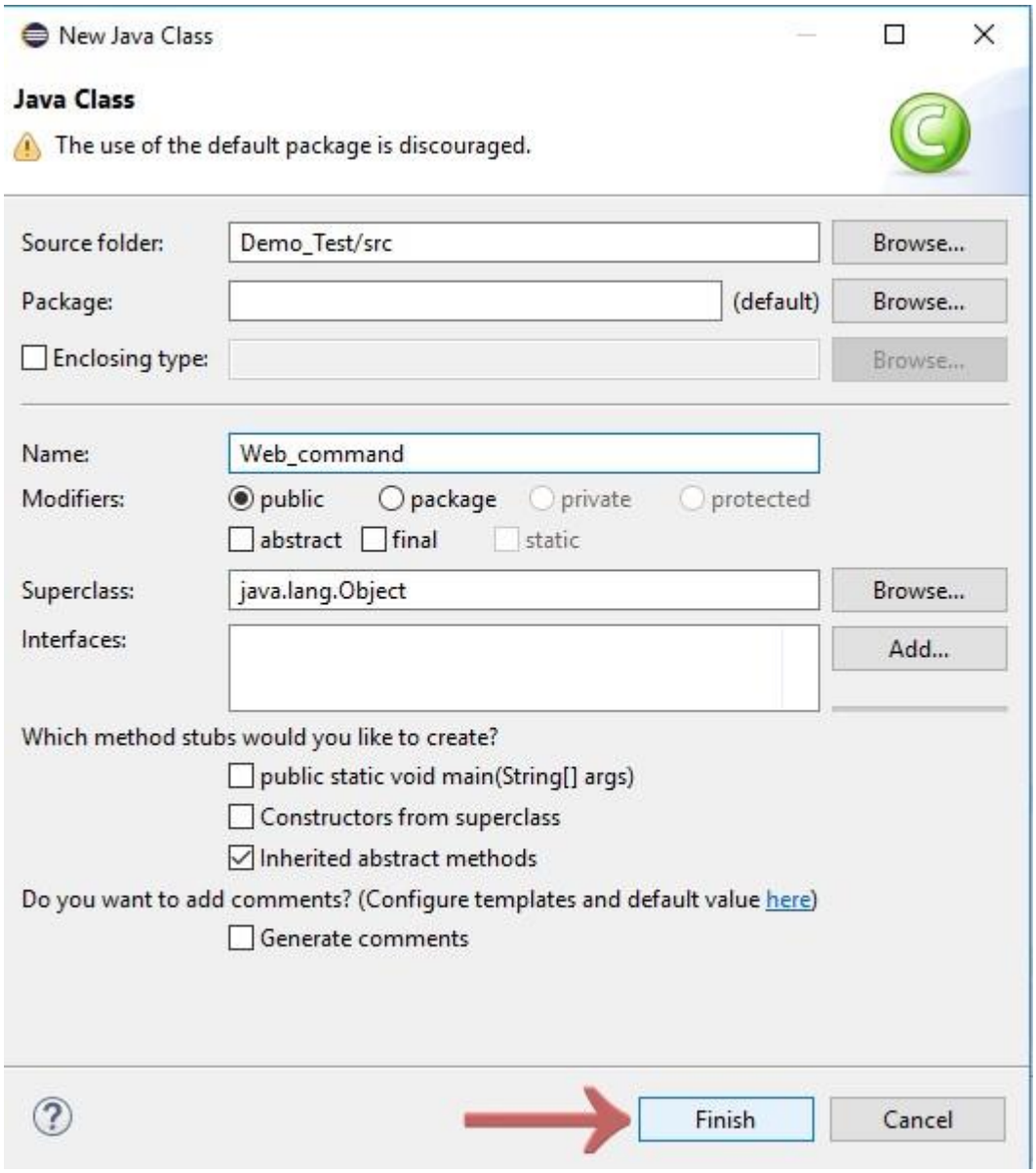
For our test purpose, we are using the home page of "Google" search engine.

We will create our test case step by step to give you a complete understanding on how to use Browser Commands in WebDriver.

- **Step1.** Launch Eclipse IDE and open the existing test suite "Demo_Test" which we have created in WebDriver Installation section of WebDriver tutorial.
- **Step2.** Right click on the "src" folder and create a new Class File from **New > Class**.



Give your Class name as "Navigation_command" and click on "Finish" button.



Step3. Let's get to the coding ground.

To automate our test scenarios, first you need to know "How to invoke/launch web browsers in WebDriver?"

Note: To invoke a browser in Selenium, we have to download an executable file specific to that browser. For example, Chrome browser implements the WebDriver protocol using an executable called ChromeDriver.exe. These executable files start a server on your system which in turn is responsible for running your test scripts in Selenium.

We have stated the procedures and methods to run our tests on different browser in later sections of this tutorial. For reference you can go through each one of them before proceeding with the actual coding.

- Running test on Firefox
- Running test on Chrome

- Running test on Internet Explorer
- Running test on Safari

- To invoke Google Chrome browser, we need to download the ChromeDriver.exe file and set the system property to the path of your ChromeDriver.exe file. We have already discussed this in earlier sessions of this tutorial. You can also refer to "Running test on Chrome Browser" to learn how to download and set System property for Chrome driver. Here is the sample code to set system property for Chrome driver:

```
// System Property for Chrome Driver
System.setProperty("webdriver.chrome.driver","D:\\ChromeDriver\\chromedriver.exe");
```

After that we have to initialize Chrome driver using ChromeDriver class.
Here is the sample code to initialize Chrome driver using ChromeDriver class:

```
// Instantiate a ChromeDriver class.
WebDriver driver=new ChromeDriver();
```

Combining both of the above code blocks, we will get the code snippet to launch Google Chrome browser.

```
// System Property for Chrome Driver
System.setProperty("webdriver.chrome.driver","D:\\ChromeDriver\\chromedriver.exe");

// Instantiate a ChromeDriver class.
WebDriver driver=new ChromeDriver();
```

- To automate our second test scenario i.e. "Get Page Title name and Title length", we have to store the title name and length in string and int variable respectively. Here is the sample code to do that:

```
// Storing Title name in the String variable
String title = driver.getTitle();
// Storing Title length in the Int variable   int
titleLength = driver.getTitle().length();
```

To print page Title name and Title length in the Console window, follow the given code snippet:

```
// Printing Title & Title length in the Console window
System.out.println("Title of the page is : " + title);   System.out.println("Length of the title is : "+ titleLength);
```

- The next test scenario requires fetching the URL and verifying it against the actual URL.

First, we will store the current URL in a String variable:

```
// Storing URL in String variable
String actualUrl = driver.getCurrentUrl();
```

Verifying the current URL as the actual URL:

```
if (actualUrl.equals("https://www.google.co.in"))
{
System.out.println("Verification Successful - The correct Url is opened.");
}
Else
{
System.out.println("Verification Failed - An incorrect Url is opened.");
}
```

- To automate our 6th test scenario (Get page Source and Page Source length), we will store the page source and page source length in the *string* and *int* variable respectively.

```
// Storing Page Source in String variable
String pageSource = driver.getPageSource();
// Storing Page Source length in Int variable   int
pageSourceLength = pageSource.length();
```

To print the length of the Page source on console window, follow the given code snippet:

```
// Printing length of the Page Source on console
System.out.println("Total length of the Page Source is : " + pageSourceLength);
```

- Finally, the given code snippet will terminate the process and close the browser. driver.close();

Combining all of the above code blocks together, we will get the required source code to execute our test script "Web_command".

The final test script will appear something like this:

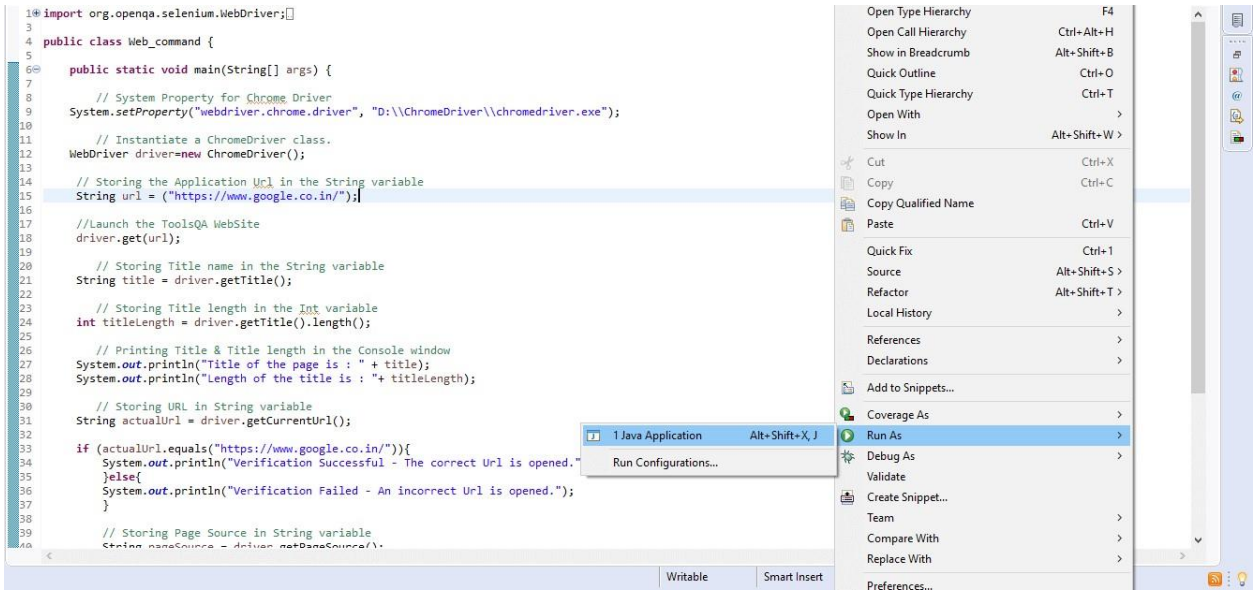
(We have embedded comment in each section to explain the steps clearly)

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

public class Web_command {

public static void main(String[] args) {

// System Property for Chrome Driver
System.setProperty("webdriver.chrome.driver","D:\\ChromeDriver\\chromedriver.exe");
```

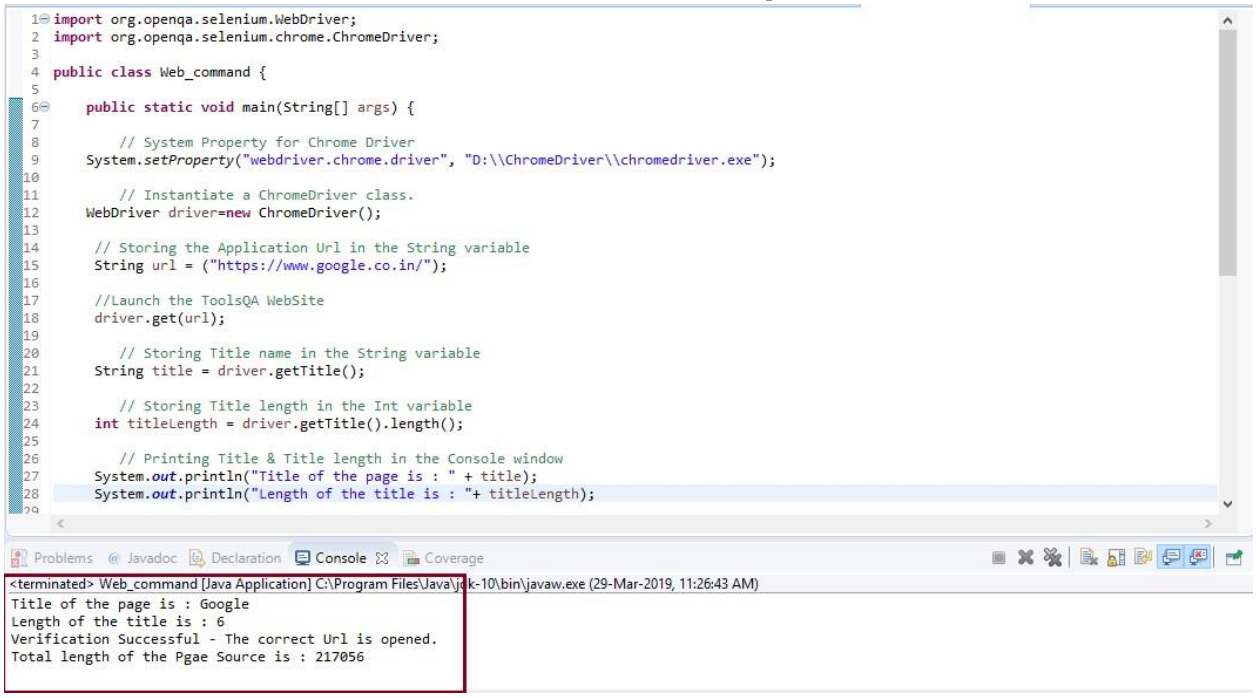
```
// Instantiate a ChromeDriver class.
WebDriver driver=new ChromeDriver();

// Storing the Application Url in the String variable
String url = ("https://www.google.co.in/");

//Launch the ToolsQA WebSite
driver.get(url);

// Storing Title name in the String variable
String title = driver.getTitle();

// Storing Title length in the Int variable
int titleLength = driver.getTitle().length();

// Printing Title & Title length in the Console window
System.out.println("Title of the page is : " + title);
System.out.println("Length of the title is : "+ titleLength);

// Storing URL in String variable
String actualUrl = driver.getCurrentUrl();

if (actualUrl.equals("https://www.google.co.in/")){
System.out.println("Verification Successful - The correct Url is opened.");
}
else{
System.out.println("Verification Failed - An incorrect Url is opened.");
      }
// Storing Page Source in String variable
String pageSource = driver.getPageSource();

// Storing Page Source length in Int variable
int pageSourceLength = pageSource.length();

// Printing length of the Page Source on console
System.out.println("Total length of the Pgae Source is : " + pageSourceLength);
//Closing browser
 driver.close();
}
}
```

To run the test script on Eclipse window, right click on the screen and click

Run as → Java application



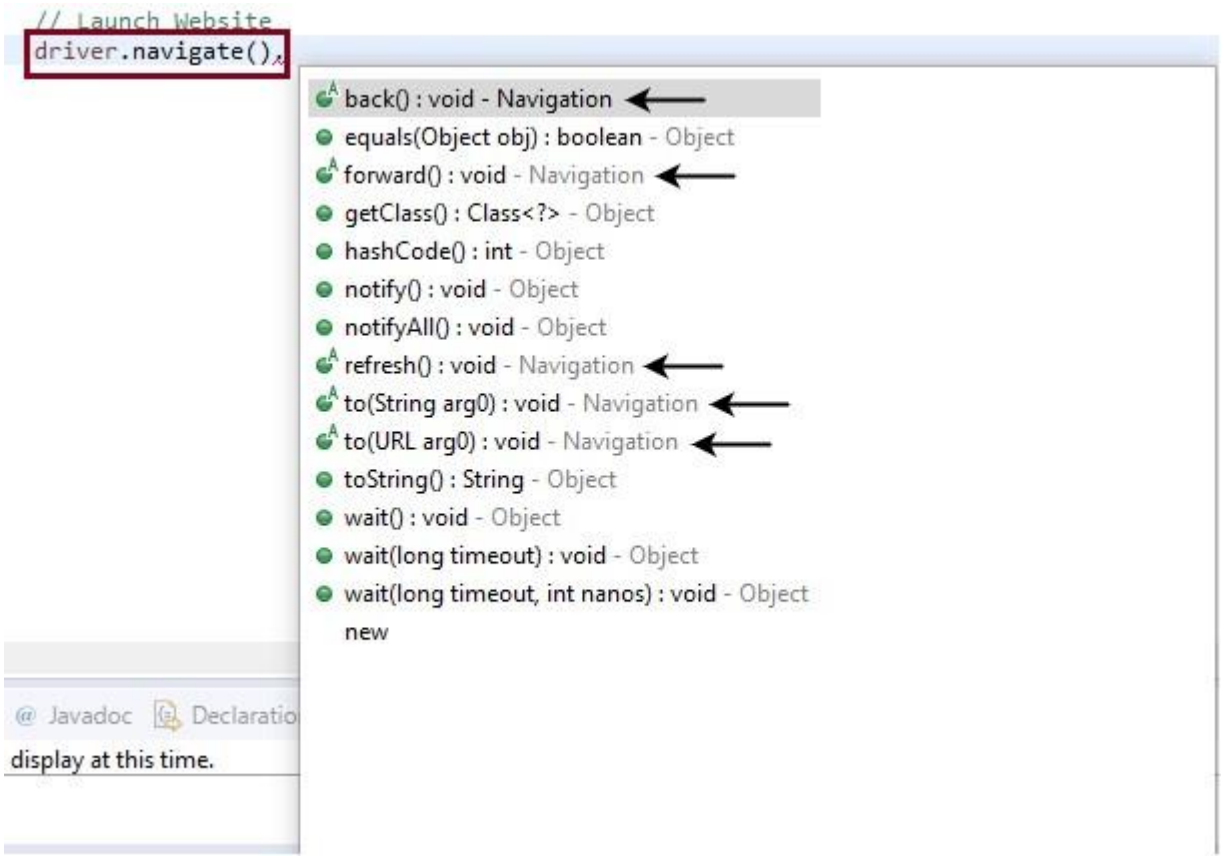
After execution, the test script will launch the chrome browser and automate all the test scenarios. The console window will show the results for print commands.



Selenium WebDriver - Navigation Commands

WebDriver provides some basic Browser Navigation Commands that allows the browser to move backwards or forwards in the browser's history.

Just like the browser methods provided by WebDriver, we can also access the navigation methods provided by WebDriver by typing driver.navigate() in the Eclipse panel.



Note: The methods having 'Navigation' as keyword are declared as Navigation commands.

Given are some of the most commonly used Browser Navigation commands for Selenium WebDriver.

1. Navigate To Command

Method: to(String arg0) : **void**

In WebDriver, this method loads a new web page in the existing browser window. It accepts *String* as parameter and returns *void*. The respective command to load/navigate a new web page can be written as:

```
driver.navigate().to("www.javatpoint.com");
```

Note: The get command (driver.get(URL);) which lies in the browser commands section does the same function as the navigate command

(driver.navigate().to("www.javatpoint.com");

2. Forward Command

Method: to(String arg0) : **void**

In WebDriver, this method enables the web browser to click on the **forward** button in the existing browser window. It neither accepts anything nor returns anything.

The respective command that takes you forward by one page on the browser's history can be written as:

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

3. Back Command

Method: back() : **void**

In WebDriver, this method enables the web browser to click on the **back** button in the existing browser window. It neither accepts anything nor returns anything.

The respective command that takes you back by one page on the browser's history can be written as: `driver.navigate().back();`

4. Refresh Command

Method: refresh() : **void**

In WebDriver, this method refresh/reloads the current web page in the existing browser window. It neither accepts anything nor returns anything.

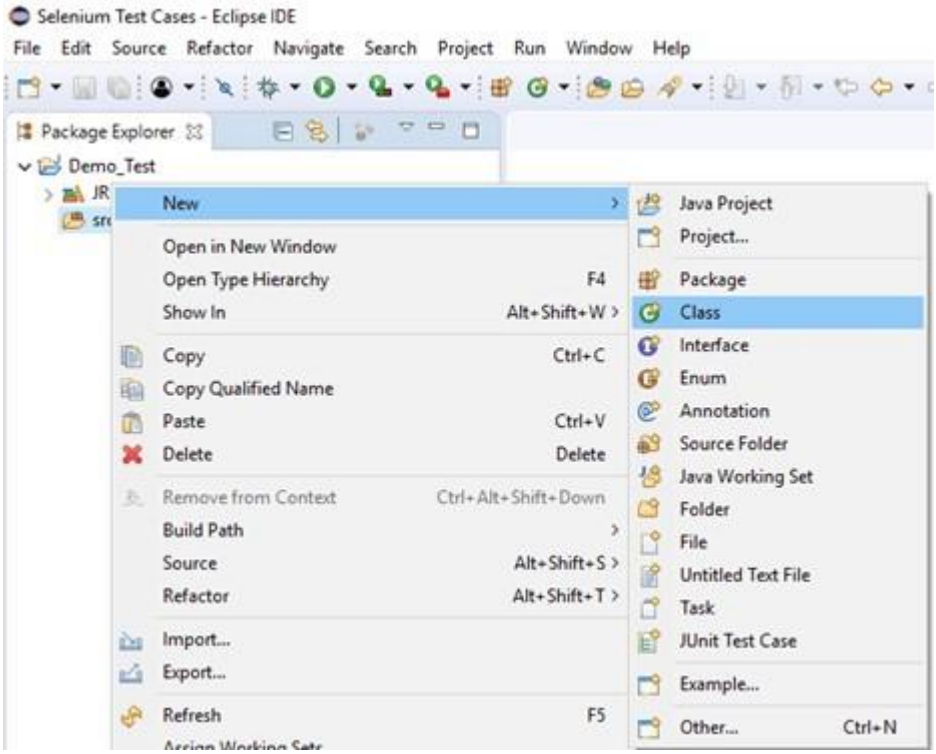
The respective command that takes you back by one page on the browser's history can be written as: `driver.navigate().refresh();`

Let us consider a sample test script which will cover most of the Navigation Commands provided by WebDriver. **In this sample test, we will automate the following test scenarios:**

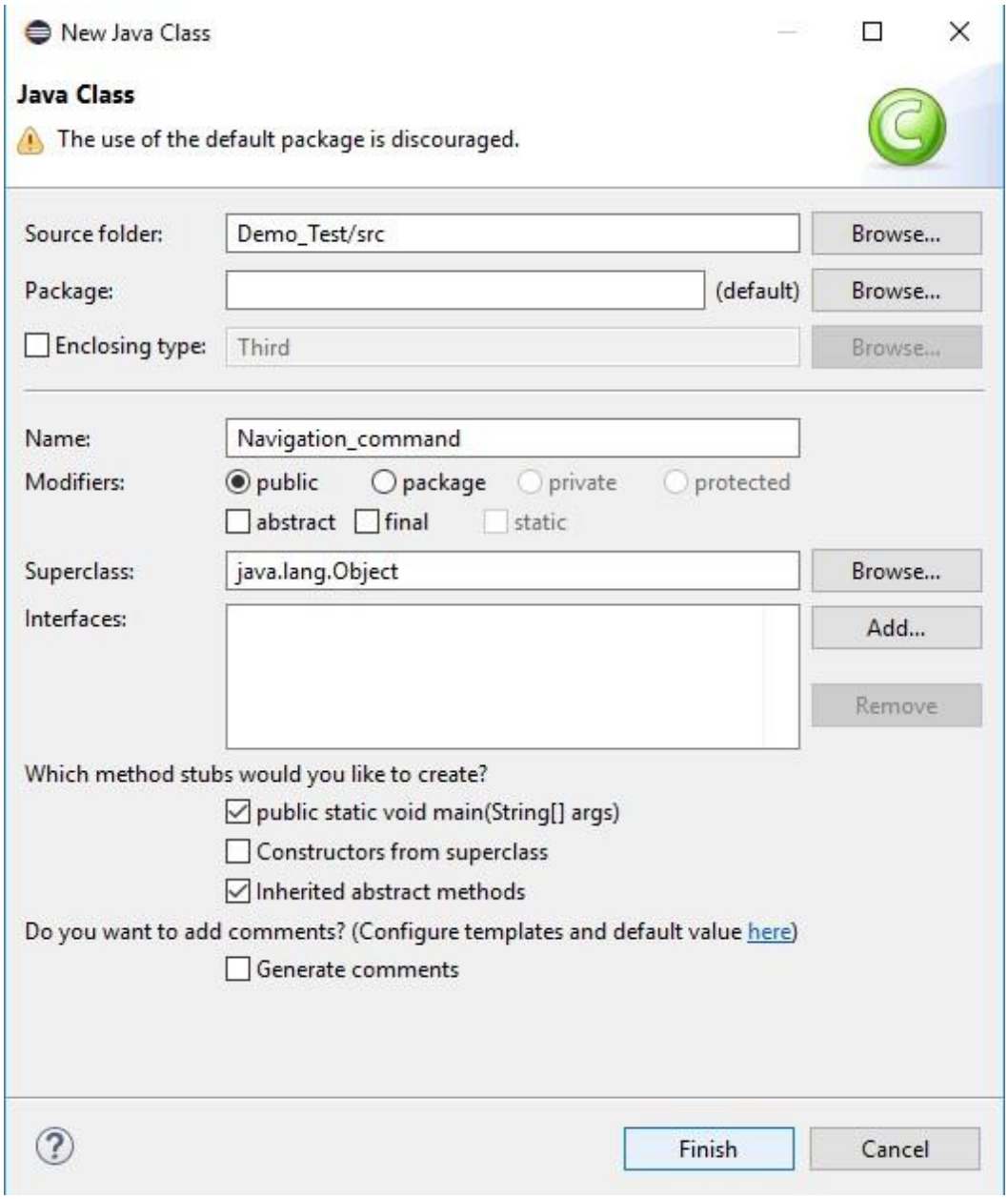
- Invoke Firefox Browser
- Navigate to URL: https://www.testandquiz.com/selenium/testing.html
- Click on the "This is a link" link (This link will redirect you to the javaTpoint website)
- Come back to the Home page using the **back** command
- Again go back to the javaTpoint website using **forward** command
- Again come back to the Home page using **To** command
- Refresh the Browser using **Refresh** command
- Close the Browser

For our test purpose, we are using a dummy web page under the URL: https://www.testandquiz.com/selenium/testing.html (You can also use this dummy web page for your Selenium Test Practices)

- **Step1.** Launch Eclipse IDE and open the existing test suite "Demo_Test" which we have created in WebDriver Installation section of WebDriver tutorial.
- **Step2.** Right click on the "src" folder and create a new Class File from **New >Class**.



Give your Class name as "Navigation_command" and click on "Finish" button.



Step3. Let's get to the coding ground.

- To invoke Firefox browser, we need to download Gecko driver and set the system property for Gecko driver. Here is the sample code to set system property for Gecko driver:

```
// System Property for Gecko Driver
System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" )
```

After that we have to initialize Gecko Driver using Desired Capabilities Class.

Here is the sample code to initialize gecko driver using DesiredCapabilities class.

```
// Initialize Gecko Driver using Desired Capabilities Class
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
capabilities.setCapability("marionette",true);
WebDriver driver= new FirefoxDriver(capabilities);
```

Combining both of the above code blocks, we will get the code snippet to launch Firefox browser.

```
// System Property for Gecko Driver
System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );
// Initialize Gecko Driver using Desired Capabilities Class
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
capabilities.setCapability("marionette",true);
WebDriver driver= new FirefoxDriver(capabilities);
```

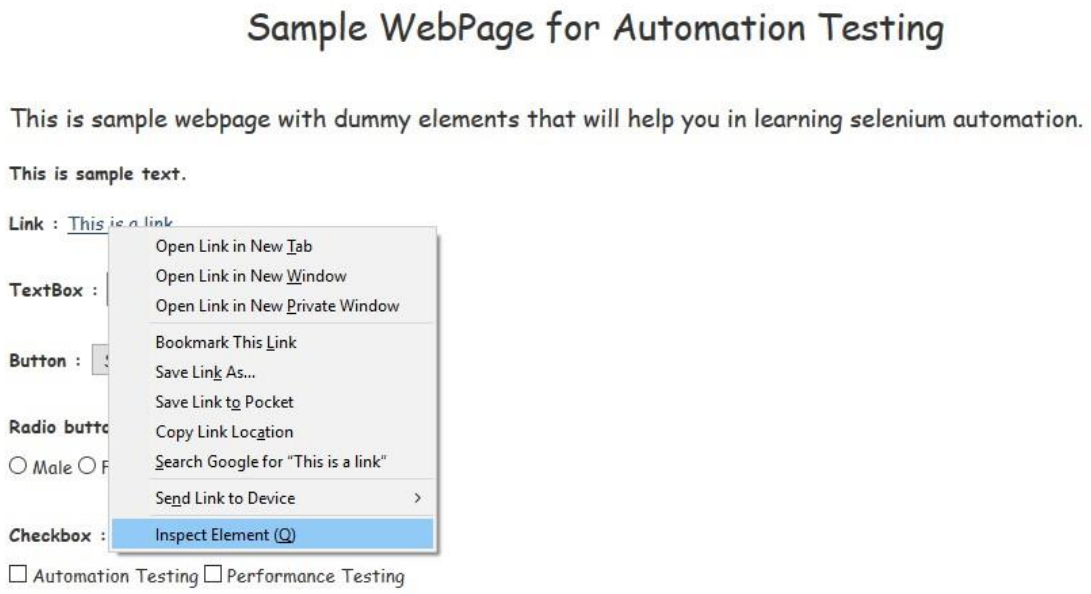
- After that we need to write the code which will automate our second test scenario (get the desired URL)

Here is the sample code to navigate to the desired URL:

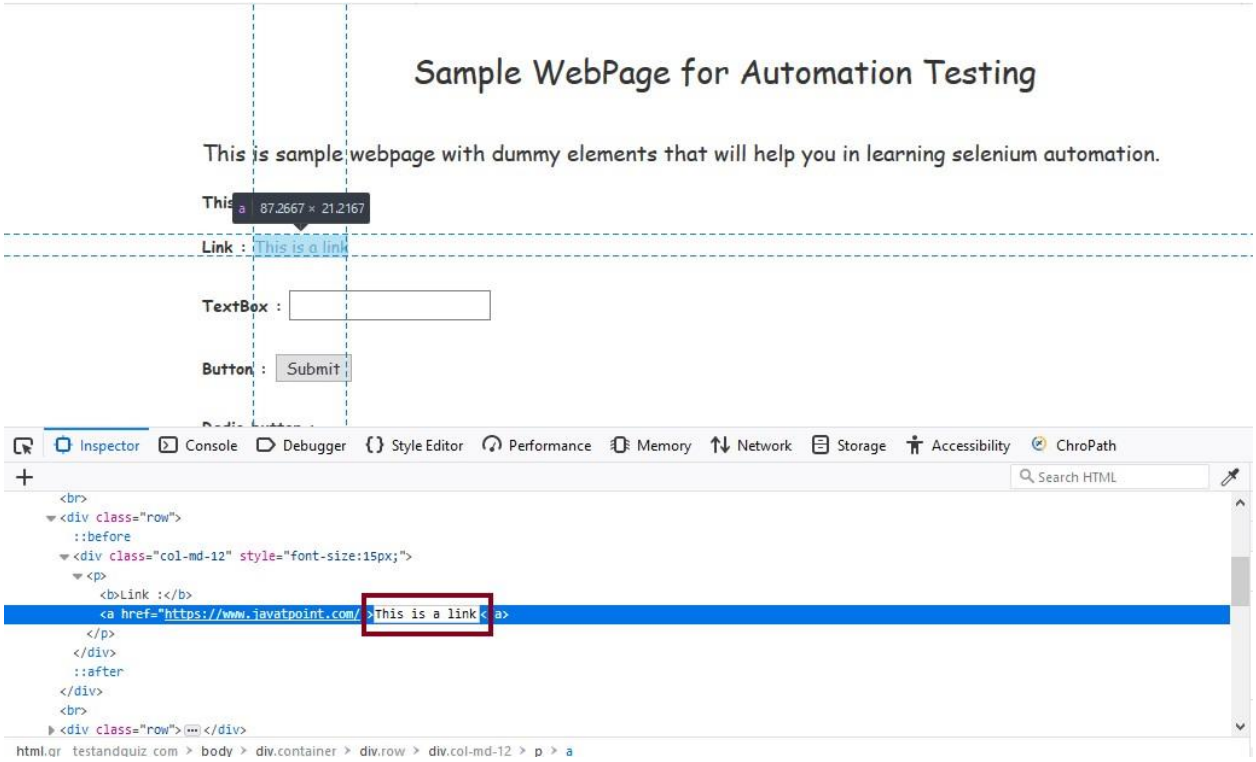
```
//Navigate to the desired URL
driver.navigate().to("https://www.testandquiz.com/selenium/testing.html");
```

- To automate our third test scenario, first we have to uniquely identify the "This is a link" link on the dummy test page. The method for finding a unique identification element involves inspection of HTML codes. Open URL: https://www.testandquiz.com/selenium/testing.html in your firefox browser.

Right click on the "This is a link" link text and select Inspect Element.



It will launch a window containing all the specific codes involved in the development of the "This is a link" link. Select the name of the link text from inspector text box.



The Java Syntax for uniquely identifying a web element through its Link Text is written as: driver.findElement(By.linkText (<linktext>)

Therefore, for locating the Link Text on the sample web page we will use the value of its Link Text: driver.findElement(By.linkText (<"This is a Link">)) Now, we need to write the code which will click on the Link Text. Here is the sample code to click on the Link Text. // Click on the Link Text using click() command driver.findElement(By.linkText("This is a Link")).click();

On click, the link will redirect the browser window to the official web page of javaTpoint website.

- To automate our fourth test scenario, we have to revert the action performed by our third test scenario. To do that, we will use the **Back** command to undo the action performed on click of the link text.

Here is the sample code to return to the home page after being directed to the javaTpoint website.

// Go back to Home Page driver.navigate().back();

- Now, the next test scenario requires us to again go to the action performed by our third test scenario i.e., the window will again directed to the javaTpoint website. Here is the sample code to go forward again to the official web page of javaTpoint website.

```
// Go forward to Registration page   driver.navigate().forward();
```

- Now, to automate our sixth test scenario, we will require to again navigate to the home page of dummy website by using the **To command**. Here is the sample code to go back to the home page.

```
// Go back to Home page   driver.navigate().to(appUrl);
```

To refresh the browser window, use the **Refresh command** as:

```
// Refresh browser   driver.navigate().refresh();
```

Finally, the given code snippet will terminate the process and close the browser.

```
driver.close();
```

Combining all of the above code blocks together, we will get the required source code to execute our test script "Navigation_command".

The final test script will appear something like this:

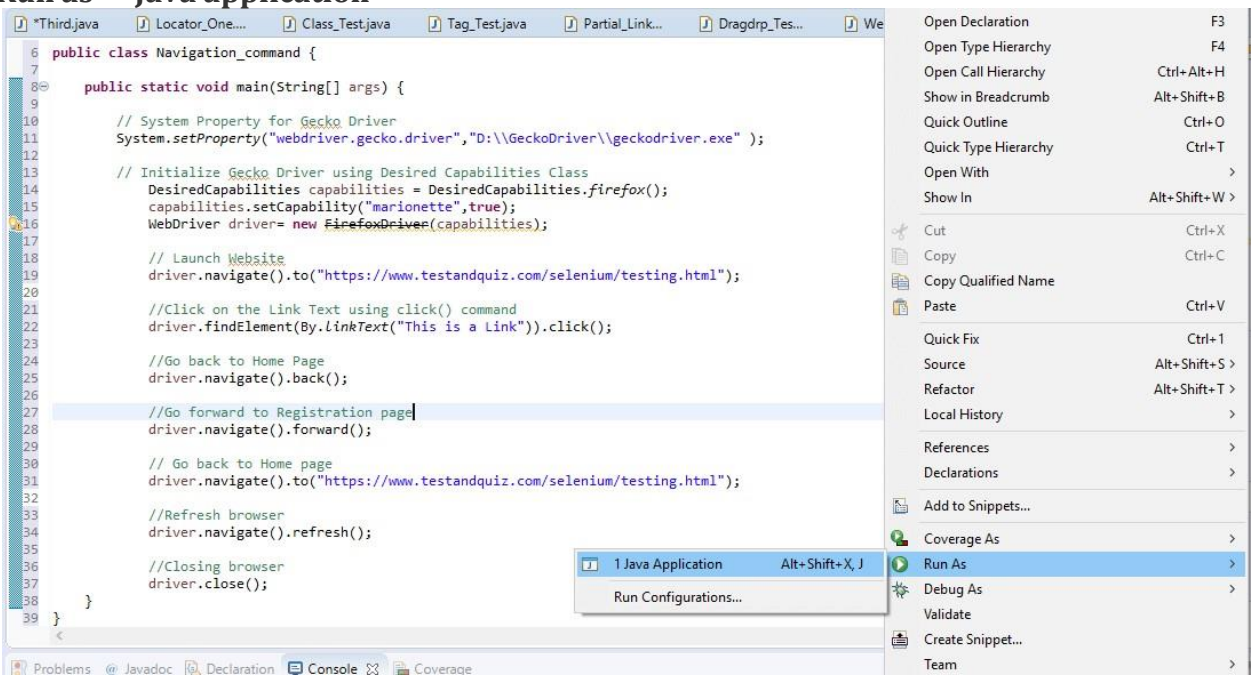
(We have embedded comment in each section to explain the steps clearly)

```
import org.openqa.selenium.By;    import
org.openqa.selenium.WebDriver;    import
org.openqa.selenium.firefox.FirefoxDriver;    import
org.openqa.selenium.remote.DesiredCapabilities;

public class Navigation_command {

  public static void main(String[] args) {

    // System Property for Gecko Driver
    System.setProperty("webdriver.gecko.driver","D:\\GeckoDriver\\geckodriver.exe" );

    // Initialize Gecko Driver using Desired Capabilities Class
       DesiredCapabilities capabilities = DesiredCapabilities.firefox();
       capabilities.setCapability("marionette",true);
WebDriver driver= new FirefoxDriver(capabilities);

       // Launch Website
       driver.navigate().to("https://www.testandquiz.com/selenium/testing.html");
```

```
        //Click on the Link Text using click() command
        driver.findElement(By.linkText("This is a Link")).click();

        //Go back to Home Page
        driver.navigate().back();

        //Go forward to Registration page
        driver.navigate().forward();

        // Go back to Home page
        driver.navigate().to("https://www.testandquiz.com/selenium/testing.html");
        //Refresh browser
        driver.navigate().refresh();
        //Closing browser
        driver.close();
    }
}
```

To run the test script on Eclipse window, right click on the screen and click

Run as → Java application



Upon execution the test script will launch the Firefox browser and automate all the test scenarios.

Handling Radio buttons

In this section, you will learn how to handle radio buttons in selenium web driver.

Following are the steps to handle the radio buttons:

Step 1: Invoke the Google Chrome browser.

The code to invoke a Google chrome browser is given below:

```
package mypack;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
public class Class1
{
   public static void main(String[] args)
   {

      System.setProperty("webdriver.chrome.driver", "C:\\work\\chromedriver.exe");
      WebDriver driver = new ChromeDriver();

   }

}
```

Step 2: The second step is to navigate to the website in which we need to handle the radio buttons.

I created the html file which contains the radio buttons. The code is given below:

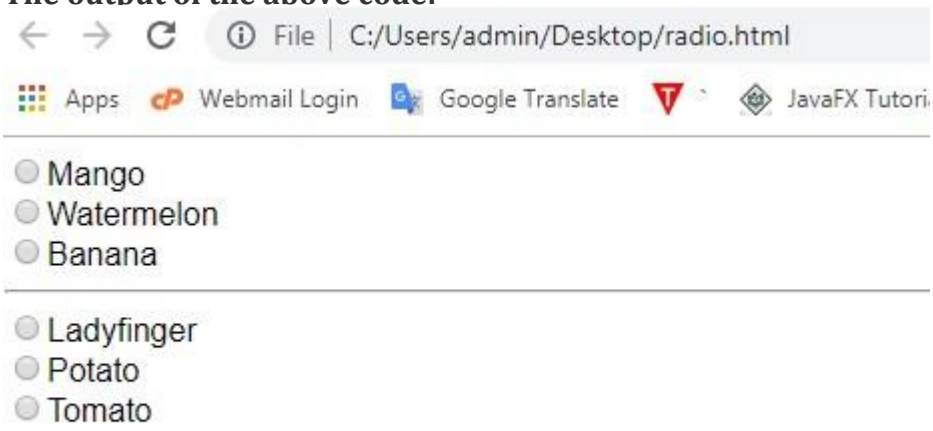
```
<html>
<head>
</head>
<body>
<input type="radio" name="group1" value="Mango">Mango<br>
<input type="radio" name="group1" value="Watermelon">Mango<br>
<input type="radio" name="group1" value="Banana">Mango<br>   <hr>
<input type="radio" name="group2" value="Ladyfinger">Ladyfinger<br>
<input type="radio" name="group2" value="Potato">Potato<br>
<input type="radio" name="group2" value="Tomato">Tomato<br>
</body>   </html>
```

The code for navigating to the above html file is given below:

```
package
mypack;
import  org.openqa.selenium.WebDriver;     import
org.openqa.selenium.chrome.ChromeDriver;

public class Class1 {
```

```
    public static void main(String[] args) {

        System.setProperty("webdriver.chrome.driver", "C:\\work\\chromedriver.exe");
        WebDriver driver = new ChromeDriver();
        driver.get("file:///C:/Users/admin/Desktop/radio.html");
        }

}
```

The output of the above code:



Step 3: Select the option Banana. We will locate the Banana radio button by inspecting its HTML codes.

There are two ways of handling the radio buttons:

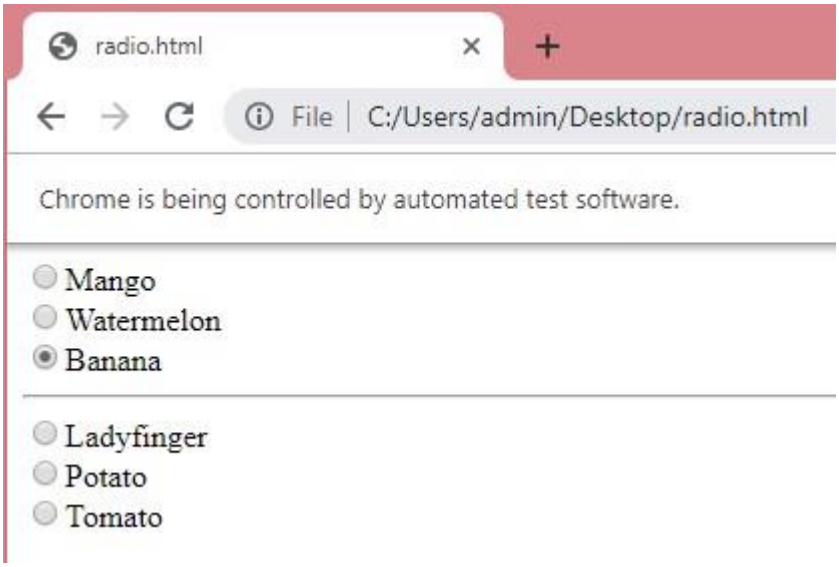
- **By using the customized path:**

The code shown below handles the radio button by using the customized path.

```
package mypack;
import org.openqa.selenium.By;   import
org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Class1 {
   public static void main(String[] args) {

      System.setProperty("webdriver.chrome.driver", "C:\\work\\chromedriver.exe");
WebDriver driver = new ChromeDriver();
      driver.get("file:///C:/Users/admin/Desktop/radio.html");
driver.findElement(By.xpath("//input[@value='Banana']")).click();
      }
```

}

In the above, we use custom Xpath. Radio buttons contain a unique attribute, i.e., value, so we use the value attribute to handle the radio button. **Output**



- **By handling the radio buttons dynamically.**
  - We will first calculate the number of radio buttons. The following is the line of code which calculates the number of radio buttons.

    int a = driver.findElements(By.xpath("//input [@name='group1']")).size();

    The above line of code calculates the number of radio buttons whose name is group1.
  - Now, we will handle the radio buttons by using the index of a particular radio button.

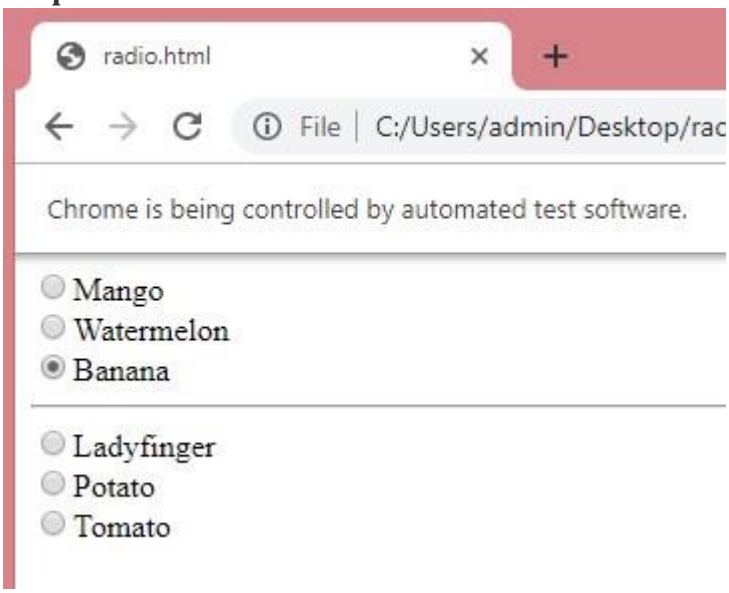
    driver.findElements(By.xpath("//input[@name='group1']")).get(2).click();

Source code

```
package mypack;
import org.openqa.selenium.By;   import
org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Class1 {
   public static void main(String[] args) {

      System.setProperty("webdriver.chrome.driver", "C:\\work\\chromedriver.exe");
WebDriver driver = new ChromeDriver();
      driver.get("file:///C:/Users/admin/Desktop/radio.html");
      int    a    =    driver.findElements(By.xpath("//input    [@name='group1']")).size();
System.out.println(a);
      for(int i=1;i<=a;i++)
      {
         driver.findElements(By.xpath("//input[@name='group1']")).get(2).click();
      }
   }}
```

In the above code, we use 'for' loop. Inside the 'for' loop, we find the third radio button of group1 by using the get(2) method.

Output



Handling Checkbox

In this section, you will learn how to handle checkbox in selenium webdriver.

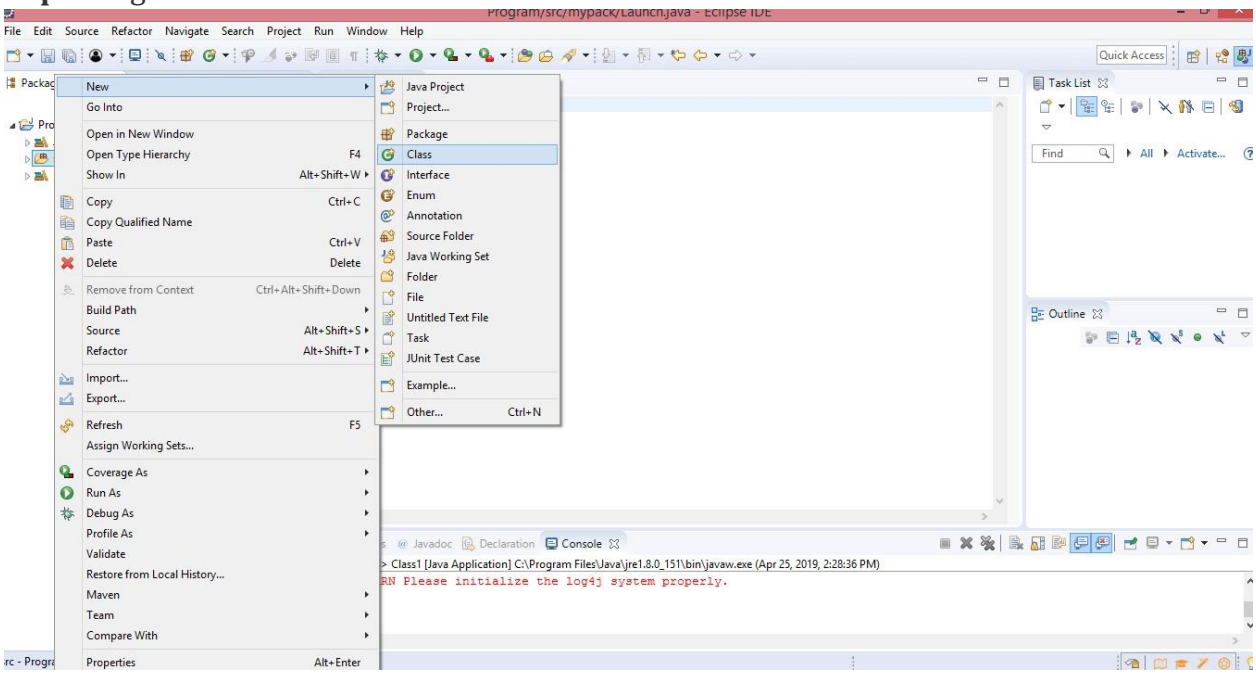
Let's create a test case in which we will automate the following scenarios:

- Invoke a Google chrome browser.
- Navigate to the website in which you handle the checkbox.
- Select the '**Senior Citizen**' checkbox from the spicejet website.
- Close the driver.

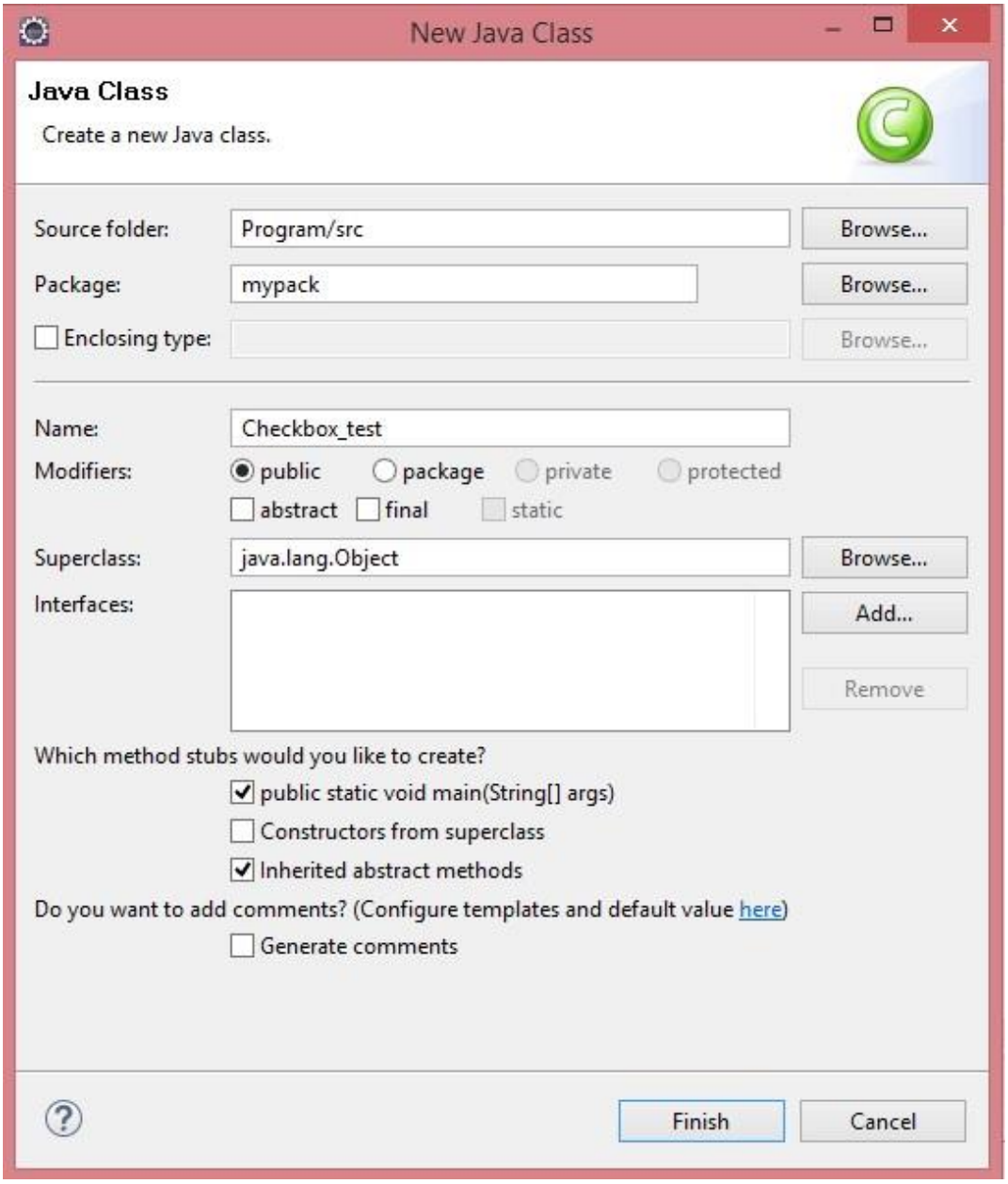
Now, we will create a test case step by step in order to provide you proper understanding of how to handle checkbox.

Step 1: Launch the Eclipse IDE.

Step 2: Right click on the src folder and then click on the New > class.



- Enter the class name. I provide the class name as Checkbox_test.



Step 3: Now, we will invoke the Google Chrome browser. We will download the chromedriver.exe file and set the system property to the path of your chromedriver.exe file. Here is the sample code to set the system property to the path of a chromedriver.exe file.

```
System.setProperty("webdriver.chrome.driver","C:\\\\work\\\\chromedriver.exe);
```

Here is the sample code to invoke the Google chrome browser.

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

Combining both of the above code blocks, we will get the code snippet to launch Google chrome browser.

```
// Set the system property
System.setProperty("webdriver.chrome.driver","C:\\\\work\\\\chromedriver.exe);

// Launch the Google Chrome browser.
WebDriver driver = new ChromeDriver();
```

Step 4: We are done with the first test case, i.e., invoking a Google chrome browser. Now we will write the code to automate the test case. Our second test case is to navigate to the "spicejet" website.

 Here is

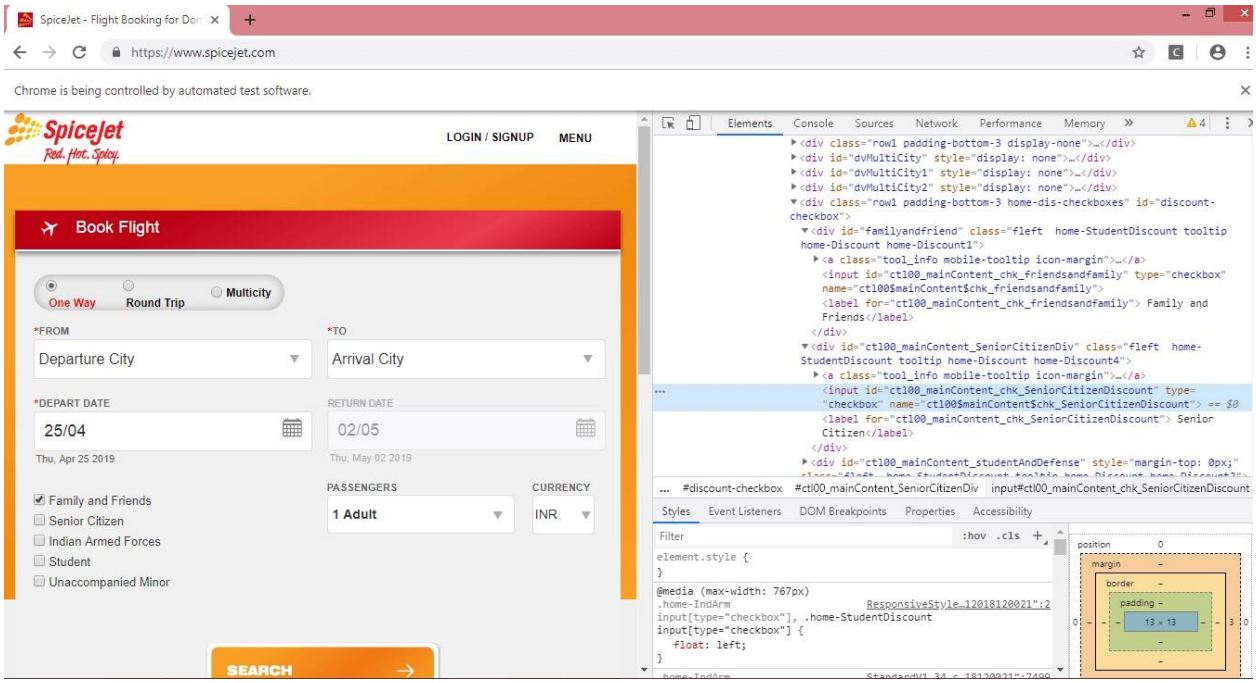
the sample code to navigate to the "spicejet" website.

```
driver.navigate().to("https://www.spicejet.com/");
```

Here is the complete code:

```
package mypack;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Checkbox_test {

  public static void main(String[] args) {
// TODO Auto-generated method stub
    System.setProperty("webdriver.chrome.driver","C:\\\\work\\\\chromedriver.exe");
    WebDriver driver = new ChromeDriver();
    driver.navigate().to("https://www.spicejet.com/");

  }}
```

Step 5: Now we try to locate the 'Senior Citizen' checkbox by inspecting its HTML code.



Note the id attribute of a checkbox.

```
<input id="ctl00_mainContent_chk_SeniorCitizenDiscount" type="checkbox"
name="ctl00$mainContent$chk_SeniorCitizenDiscount"> == $0
```

In the above case, we observe that 'id' is a unique attribute, so we locate the checkbox by using an **id attribute**.

Step 6: To automate the third test case, we need to write the code that will locate '**Senior Citizen**' checkbox.

Here is the code that will handle the "Senior Citizen" checkbox.

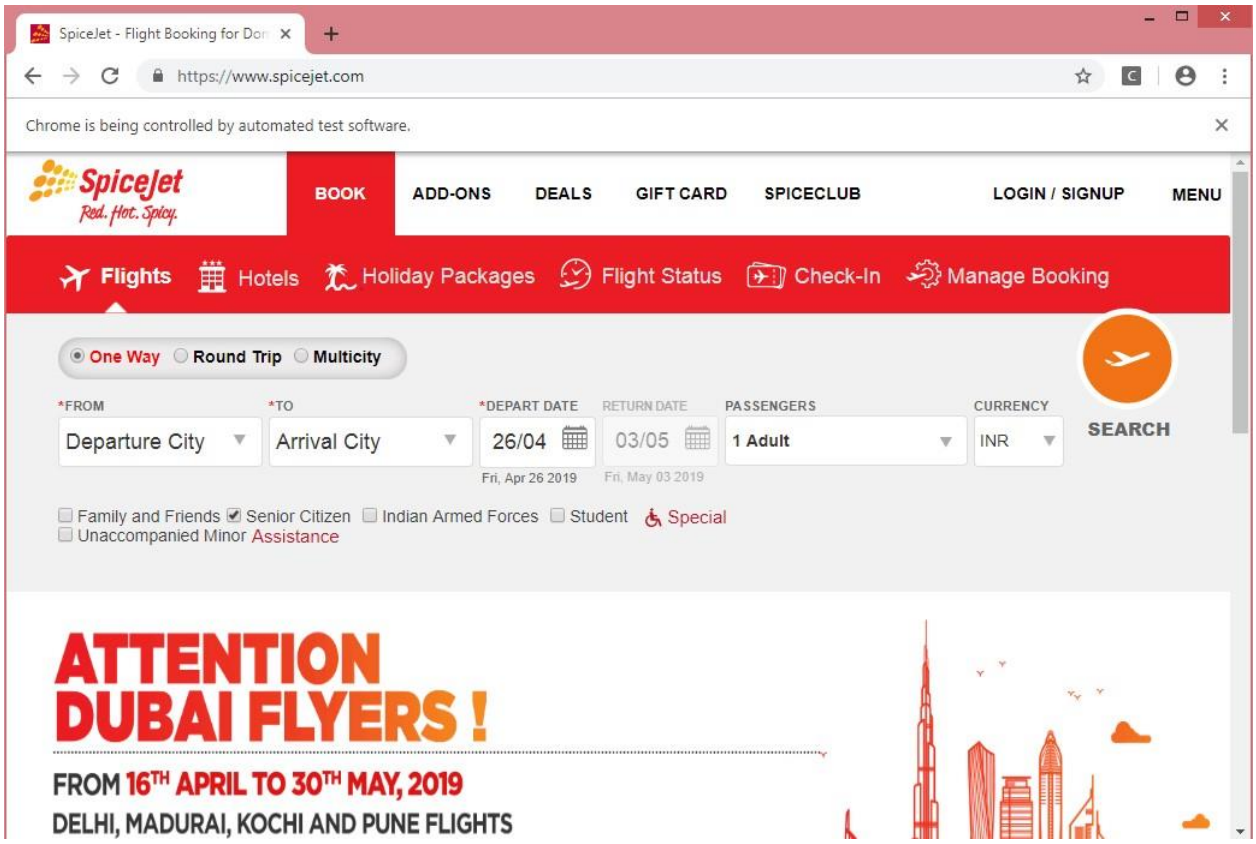
```
package mypack;
import org.openqa.selenium.By; import
org.openqa.selenium.WebDriver; import
org.openqa.selenium.chrome.ChromeDriver; public
class Checkbox_test
{

public static void main(String[] args)
{
    // TODO Auto-generated method stub
    System.setProperty("webdriver.chrome.driver","C:\\\\work\\\\chromedriver.exe");
    WebDriver driver = new ChromeDriver();
driver.navigate().to("https://www.spicejet.com/");
```

```
System.out.println(driver.findElement(By.cssSelector("input[id*='SeniorCitizenDiscount']"
)).isSelected());
      driver.findElement(By.cssSelector("input[id*='SeniorCitizenDiscount']")).click();

System.out.println(driver.findElement(By.cssSelector("input[id*='SeniorCitizenDiscount']"
)).isSelected());

driver.close();
   }

}
```

In the above code, we haven't used the complete 'id' attribute value as it is very big. I have used the half part of the 'Senior Citizen' checkbox, and other half part is represented in the form of regular expression, i.e., '*='. We have used two methods in the above code:

- **isSelected():** This method determines whether the checkbox is selected or not. If the checkbox is selected, then this method returns true otherwise false.
- **click():** This method selects the locator. In this case, it is selecting the "Senior Citizen" checkbox.

Output



Output on the Console