SELENIUM

Selenium Fundamentals

What is Selenium?

- Selenium is a powerful open-source suite of tools and libraries designed to enable and support the automation of web browsers using best techniques to remotely control browser instances and emulate a user's interaction with the browser.
- It enables developers and testers to write test scripts in various programming languages to control web browsers, ensuring that web applications behave as expected.
- it is a toolset for web browser automation that uses the best techniques available to It is functional for all browsers, works on all major OS, and its scripts are written in various languages i.e., Python, Java, C#, etc.,

Features and uses of Selenium

Features:

- Web Based Automation Testing
- Open Source
- Platform independent - multiple language, browser and OS support
- Integrates with various testing frameworks.

Uses:

- Automated Web Automation Testing
- Cross Browser Testing
- Web Scraping
- Functional Testing

Selenium Components

- Selenium comprises of many tools each for a specific purpose. Among them, Selenium WebDriver, Selenium IDE and Selenium Grid are most used components.

Component	Functionality
Selenium WebDriver	It is a simple and concise programming interface that drives browsers effectively as a user would
Selenium IDE	IDE is an easy-to-use browser extension that records a user's actions in the browser using existing Selenium commands
Selenium Grid	Enables parallel execution of tests on multiple machines and browsers simultaneously.

Prerequisites to understand browser interactions

Listed below are a few key concepts which need to be understood before starting to work with Selenium WebDriver

- Basics of Web Technology
- Browser DOM
- Developer Tools for Browsers
- WebElements, Attributes and methods

Basics of Web Technology

- Web technology refers to the collection of tools, protocols, languages, and methodologies that enable the creation, delivery, and interaction with content over the World Wide Web.
- Web technology can be broadly classified into three main categories: client-side, server-side, and database technologies.
- Client-side/frontend technologies are responsible for creating the visual and interactive aspects of a website or web application. HTML, CSS and JavaScript are the languages used for this.
- Inorder to write Selenium scripts that automate web browser interactions, it is important to understand syntax of frontend technologies HTML and CSS which build the web pages that we are trying to interact with.

Components of Web - Web pages, web sites, web server and web browser

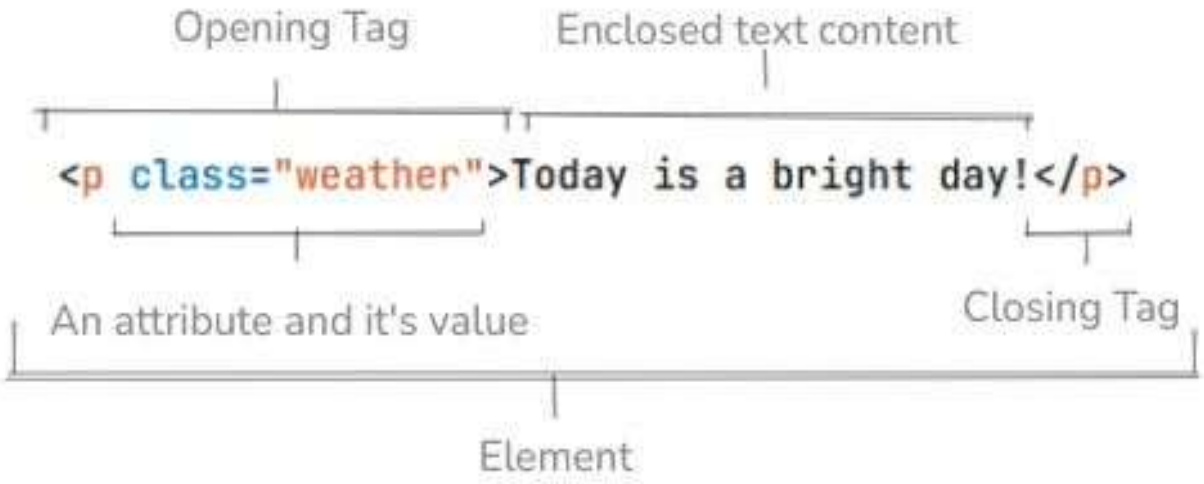
- A web page is a simple document displayable by a web browser. A web page can embed a variety of different types of resources such as:
  - Style information(CSS) — controlling a page's look-and-feel.
  - Scripts(HTML) — which add interactivity to the page, images, sounds, and videos..
  - Functionality/Behaviour(JavaScript)
- WebSite is a collection of web pages grouped together into a single resource, with links connecting them together that share a unique domain name
- A web browser is a piece of software that retrieves and displays web pages. It serves as the gateway to the internet, enabling users to access, explore, and interact with a myriad of websites, applications, and online content.
- Web Server is a computer that hosts a website on the internet.

HTML

- **HTML** (HyperText Markup Language) is the most basic building block of the Web. It defines the layout of a webpage using elements, attributes and tags, allowing for the display of text, images, links, and multimedia content.
- Hypertext refers to links that connect web pages to one another, either within a single website or between websites.
- HTML uses "markup" to annotate text, images, and other content for display in a Web browser.
- An HTML element is set off from other text in a document by "tags", which consist of the element name surrounded by < and >. The name of an element inside a tag is case-insensitive.
- For example, the <head> tag can be written as <Head>, <HEAD>, or in any other way.
- Basic HTML tags that structure the page are <html>, <head>, <body> and <title>

HTML elements and tags

- A typical HTML element includes an opening tag with some attributes, enclosed text content, and a closing tag.
- Elements and tags are not the same things. Tags begin or end an element in source code, whereas elements are part of the DOM.



HTML Attributes and DOM

- Attributes contain extra information about the element that you don't want to appear in the actual content.
- Here, **class** is the ***attribute name*** and **weather** is the ***attribute value***.
- An attribute should always have the following:
  - A space between it and the element name (or the previous attribute, if the element already has one or more attributes).
  - The attribute name followed by an equal sign.
  - The attribute value wrapped by opening and closing quotation marks.
- DOM (Document Object Model):
- The DOM is a programming interface for web documents. It represents the document as nodes and objects, allowing languages like JavaScript to manipulate the content, structure, and style of the document.
- Each element in the DOM is a node. HTML tags are represented as elements in the DOM

Nested HTML elements

- You can put elements inside other elements too — this is called **nesting**.
- If we wanted to state that our cat is very grumpy, we could wrap the word "bright" in a element, which means that the word is to be strongly emphasized (bold):

```
<p> Today is a <strong>bright</strong> day! </p>
```

- You do however need to make sure that your elements are properly nested.
- In the example above, we opened the <p> element first, then the element; therefore, we have to close the element first, then the <p> element.
- **The following is incorrect:**

```
<p> Today is a <strong>bright day! </p></strong>
```

Empty HTML Elements

- Some elements have no content and are called **empty elements**.
- Take the <img> element that we already have in our HTML page:

```
<img src="images/puppies.png" alt="Here are some puppies!">
```

- This contains two attributes, but there is no closing </img> tag and no inner content.
- This is because an image element doesn't wrap content to affect it. Its purpose is to embed an image in the HTML page in the place it appears.

Anatomy of a HTML Document

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My test page</title>
  </head>
  <body>
    <img src="images/puppies.png" alt="Here are some puppies!">
  </body>
</html>
```

HTML Image element

```
<img src="images/puppies.png" alt="Here are some puppies!">
```

- As we said before, it embeds an image into our page in the position it appears. It does this via the src (source) attribute, which contains the path to our image file.
- We have also included an alt (alternative) attribute. In this attribute, you specify descriptive text for users who cannot see the image, possibly because of the following reasons:
  - They are visually impaired. Users with significant visual impairments often use tools called screen readers to read out the alt text to them.
  - Something has gone wrong causing the image not to display.

HTML elements used to markup text

- **Headings -** Heading elements allow you to specify that certain parts of your content are headings — or subheadings.
- HTML contains 6 heading levels, <h1>–<h6>, although you'll commonly only use 3 to 4 at most:

```
<h1>My main title</h1>
<h2>My top level heading</h2>
<h3>My subheading</h3>
<h4>My sub-subheading</h4>
```

Paragraphs & Lists

- As explained in previous slides, <p> elements are for containing paragraphs of text; you'll use these frequently when marking up regular text content:

```
<p>This is a paragraph</p>
```

Lists: A lot of the web's content is lists and HTML has special elements for these.

- Marking up lists always consist of at least 2 elements.
- The most common list types are ordered and unordered lists:
- **Unordered lists** are for lists where the order of the items doesn't matter, such as a shopping list. These are wrapped in a <ul> element.
- **Ordered lists** are for lists where the order of the items does matter, such as a recipe. These are wrapped in an <ol> element.

Lists

- Each item inside the lists is put inside an <li> (list item) element.
- For example, if we wanted to turn the part of the following paragraph into a list:

```
<p>The basket has apples, oranges and grapes.</p>
```

We could modify the markup to this:

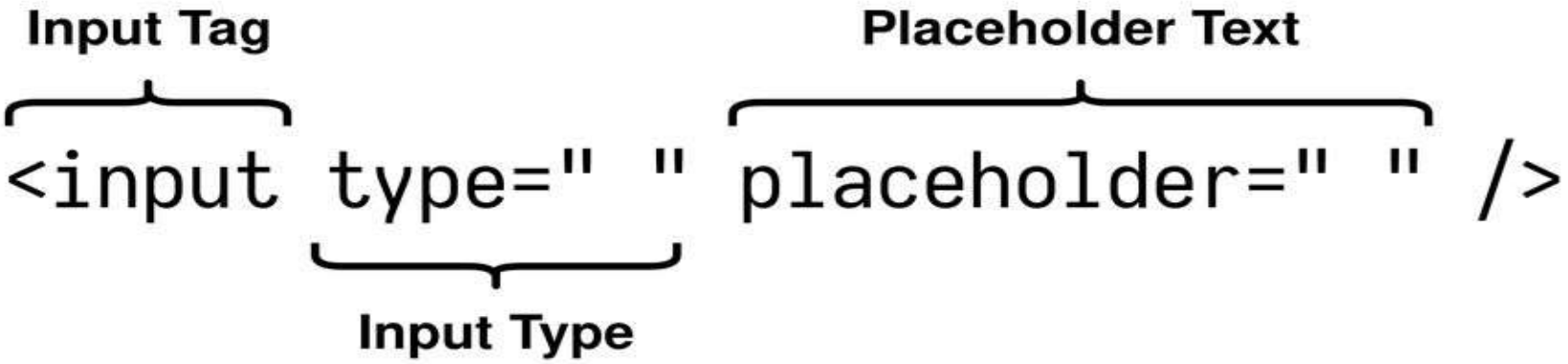
```
<p>The basket has: </p>
<ul>
  <li>apples</li>
  <li>oranges</li>
  <li>grapes</li>
</ul>
```

Links

- Links are very important — they are what makes the web a web!
- To add a link, we need to use a simple element — <a> — "a" being the short form for "anchor".
- A link declaration looks as shown below:

Inputs

- Getting user inputs is one of the most fundamental operations that you can do on a webpage.
- The base syntax for declaring an input is:



Input Types

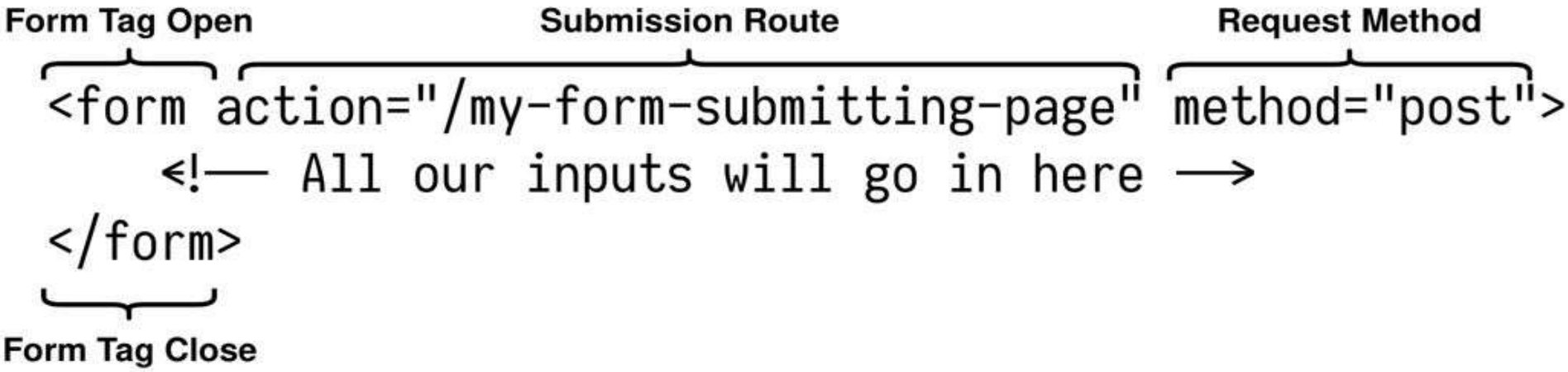
HTML code:

```
<input type="text" placeholder="Placeholder Text">

<input type="date">

<input type="color" placeholder="#ff0000">

<input type="file">

<input type="checkbox">

<input type="radio">
```

Output:

Forms

- Forms are a common element to encounter on the web.
- Declaring a form is simple, let's take a look at the syntax:



Forms

```
<form action="/my-form-submitting-page" method="post">
    <!-- All our inputs will go in here -->
</form>
```

- action - the URL to send form data to
- method - the type of HTTP request

Example of a simple form

```
<h1>Sign In</h1>

<form action="/sign-in-url" method="post">
    <input type="text" placeholder="Username">
    <input type="password" placeholder="Password">
    <button>Login</button>
</form>
```

Simple Validations

- The 'required' attribute validates that an input is not empty
- There are also type validations.
- Try changing "type" from "text" to "email"

```
<h1>Sign In</h1>

<form action="/sign-in-url" method="post">
    <input type="text" placeholder="Username" required>
    <input type="password" placeholder="Password" required>
    <button>Login</button>
</form>
```

CSS

- **Cascading Style Sheets** (**CSS**) is a stylesheet language used to describe the presentation of a document written in HTML or XML.
- CSS describes how elements should be rendered on screen, on paper, in speech, or on other media.
- CSS is among the core languages of the **open web** and is standardized across Web browsers.
- For example, if you wanted to make all the paragraphs on a page blue, this is the CSS you would use:

```
p {
  color: blue;
}
```

Writing CSS

index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>CSS Demo Page</title>
    <style>
        p {
            color: blue;
        }
    </style>
</head>
<body>
    <h1>This is a heading!</h1>
    <p>This paragraph is blue!</p>
</body>
</html>
```

- You can write your CSS inside a <style> tag in the <head> tag.
- But, this is not really good practice.

Writing CSS

index.html

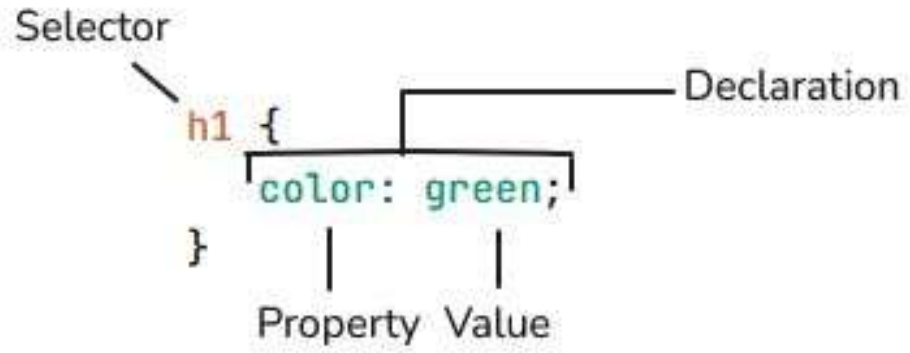
```html
<!DOCTYPE html>
<html>
<head>
    <title>CSS Demo Page</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>This is a heading!</h1>
    <p>This paragraph is blue!</p>
</body>
</html>
```

style.css

```css
p {
  color: blue;
}
```

- The best practice is to split your HTML and CSS into separate files as shown here.
- A <link> tag is used specify the relationship between the current document and an external one.

Basics of CSS ruleset



- CSS is a rule-based language — you define rules by specifying groups of styles that should be applied to particular element or groups of elements on your web page.

- **Selector:** The HTML element name at the start of the rule set. It selects the element(s) to be styled.
- **Properties**: Ways in which you can style a given HTML element. (In this case, color is a property of the <h1> elements.)
- **Property Value**: To the right of the property, after the colon, we have the property value, which chooses one out of many possible appearances for a given property (there are many color values besides green).

CSS Selectors

- A CSS selector is the first part of a CSS Rule.
- They tell the browser which HTML elements should be selected to have the CSS property values inside the rule applied to them.
- The element or elements which are selected by the selector are referred to as the subject of the selector.
- There are 4 CSS selectors that are used in general: Tag selector, ID selector, class selector and attribute selector.
- The universal selector is indicated by an asterisk (*). It selects everything in the document.

Types of CSS Selectors

Selector Name	**What Does It Select?**	**Example**
Tag Selector	All HTML element(s) of the specified type.	**p** Selects: <p>
ID Selector	The element on the page with the specified ID.	**#my-id** Selects: <p id="my-id"> and <a id="my-id">
Class Selector	The element(s) on the page with the specified class (multiple class instances can appear on a page).	**.my-class** Selects: <p class="my-class"> and <a class="my-class">
Attribute Selector	The element(s) on the page with the specified attribute.	**img[src]** Selects: <img src="myimage.png"> **but not** <img>

CSS Specificity Hierarchy

- CSS Specificity is a fundamental concept in CSS that determines the order in which styles are applied.
- If there are two or more CSS rules that point to the same element, the selector with the highest specificity will "win", and its style declaration will be applied to that HTML element.

Selector Type	**Example**	**Priority**
Inline Styles		1000
IDs	#id	100
Classes, Attributes	.classes, [attributes]	10
Element Tags	div	1

CSS pseudo classes and n^{th}-child()

- A pseudo-class is a selector that selects elements that are in a specific state.

style.css

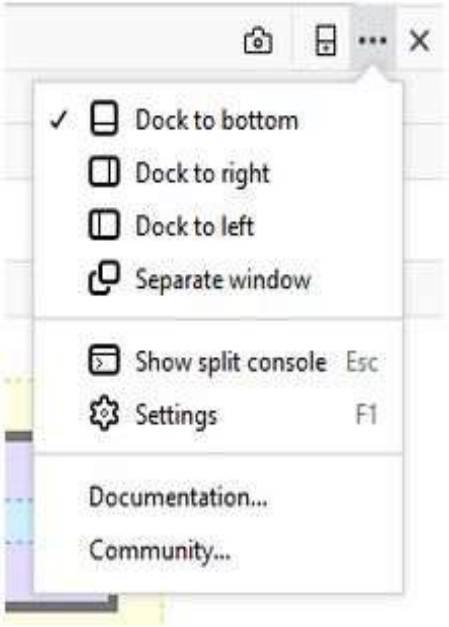
```css
li:nth-child(2) {
  color: lime;
}
/* Selects every fourth element
   among any group of siblings */
li:nth-child(4n) {
  color: pink;
}
```

index.html

```html
<ul>
    <li>One</li>
    <li>Two</li>
    <li>Three</li>
    <li>Four</li>
    <li>Five</li>
    <li>Six</li>
    <li>Seven</li>
    <li>Eight</li>
</ul>
```

Browser Developer Tools

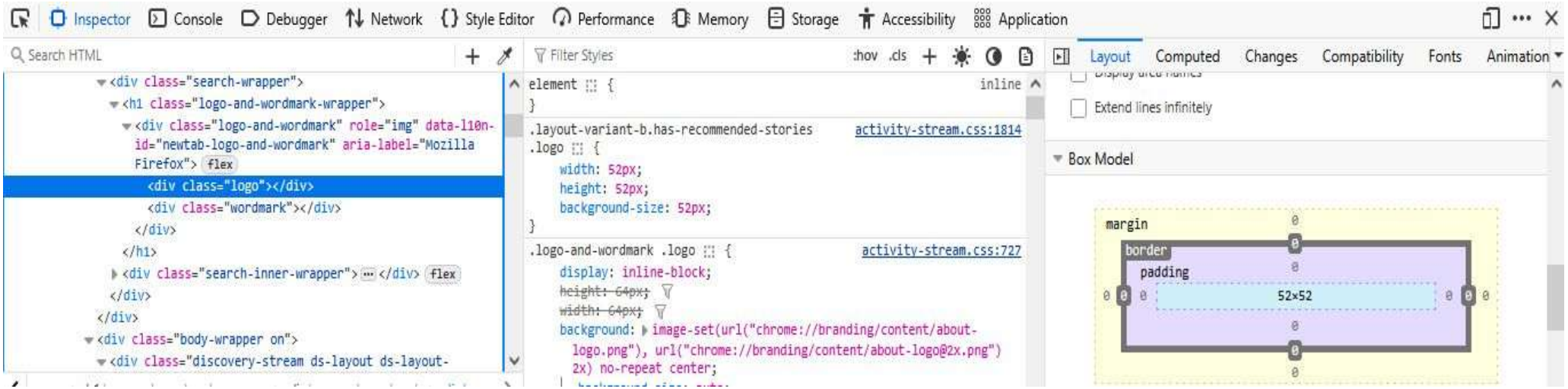
- You can open the Firefox Developer Tools from the menu by selecting *Tools* > *Web Developer* > *Toggle Tools* or use the keyboard shortcut **Ctrl + Shift + I** on Windows and Linux, or **Cmd + Opt + I** on macOS.



	This button only appears when there are multiple iframes on a page. Click it to display a list of the iframes on the current page and select the one with which you want to work.
	Click this button to take a screenshot of the current page. (**Note:** This feature is not turned on by default and must be enabled in settings before the icon will appear.)
	Toggles Responsive Design Mode.
•••	Opens the menu that includes docking options, the ability to show or hide the split console, and Developer Tools settings. The menu also includes links to the documentation for Firefox Web Tools and the Mozilla Community.
×	Closes the Developer Tools

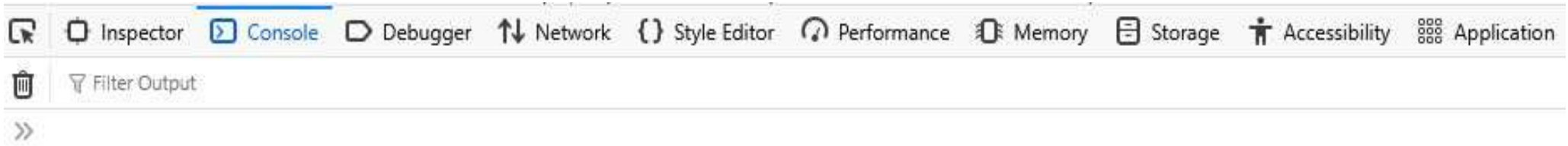
Page Inspector

- Page Inspector is a Firefox Dev Tool used to view and edit page content and layout. Visualize many aspects of the page including the box model, animations, and grid layouts.



Web Console

- Web Console is another browser developer tool that is used to see messages logged by a web page and interact with the page using JavaScript.



WebDriver

Selenium WebDriver API/Client Libraries/Language Bindings

- WebDriver API is the programming interface you interact with directly.
- Selenium provides support to multiple libraries such as Ruby, Python, Java, etc as language bindings have been developed by Selenium developers to provide compatibility for multiple languages.
- For instance, if you want to use the browser driver in Python, use the Python Bindings. You can download all the supported language bindings of your choice from the official site of Selenium.
- When you write driver.find_element() or driver.click(), you're using this API layer.

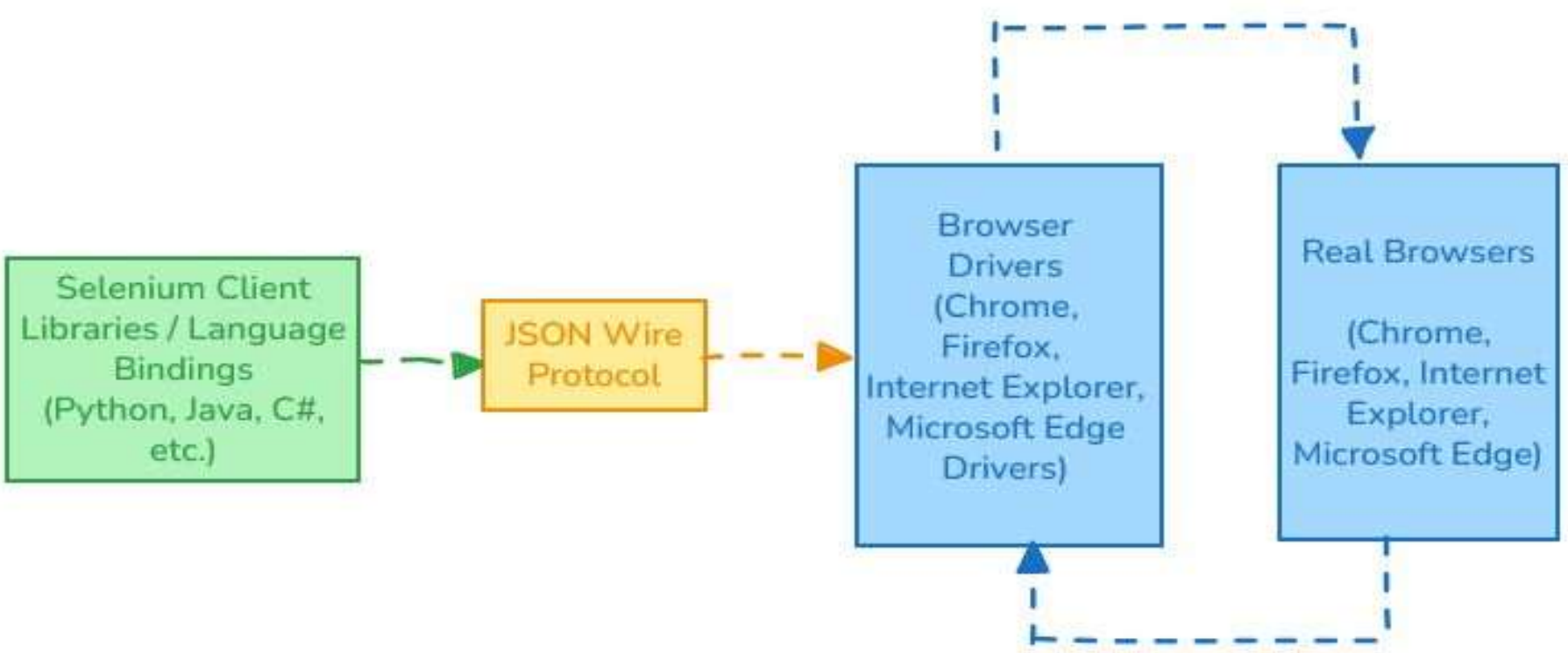
JSON Wire Protocol / W3C WebDriver Protocol

- JSON is an acronym for JavaScript Object Notation. It is an open standard that provides a transport mechanism for transferring data between client and server on the web. It provides support for various data structures like arrays and objects which makes it easier to read and write data from JSON.
- This acts as a translator between your code and the browser. Think of it as a standardized language that all browsers can understand.
- JSON serves as a REST (Representational State Transfer) API that exchanges information between HTTP servers.
- When you call a method like click(), it gets converted into HTTP requests (usually POST/GET) with JSON payloads.

Selenium WebDriver Architecture

- Selenium WebDriver Architecture is made up of four core components:

  1. WebDriver API/Client Libraries, 2. JSON Wire Protocol / W3C WebDriver Protocol 3. Browser Drivers 4. Web Browsers



Browser and Browser Drivers

Web Browsers:

- The actual browsers where tests run. Each browser has built-in automation capabilities that the drivers tap into.
- Selenium provides support for multiple browsers like Chrome, Firefox, Safari, Internet Explorer etc

Browser Drivers: These are executable files that act as intermediaries between WebDriver and specific browsers. Each browser has its own driver:

- ChromeDriver for Google Chrome
- GeckoDriver for Firefox
- EdgeDriver for Microsoft Edge
- SafariDriver for Safari

The driver knows how to communicate with its specific browser and translate WebDriver commands into browser-specific actions.

How It All Works Together

Here's the step-by-step flow when you execute a WebDriver command:

1. **You write code**: `driver.find_element(By.ID,"submit-button").click()`
2. **Client library processes**: The Python/Java/C# library converts this into a standardized command
3. **HTTP request sent**: The command becomes an HTTP request sent to the browser driver (usually running on localhost:port)
4. **Driver interprets**: ChromeDriver/GeckoDriver receives the request and understands what action to perform
5. **Browser automation**: The driver tells the actual browser to find the element and click it
6. **Response returned**: The browser performs the action and sends back a response (success/failure/element data)
7. **Your code continues**: The response travels back through the chain to your code

Browser Implementations

- Browser implementations are the browser-specific mechanisms that enable WebDriver automation. Each browser vendor (Google, Mozilla, Microsoft, Apple) implements WebDriver support differently within their browser architecture.

Browser	**Driver**
Chrome	ChromeDriver
Firefox	GeckoDriver
Internet Explorer	IEDriverServer
Safari	SafariDriver (built intomacOS)
Microsoft Edge	Edge Driver

Selenium Standalone Server

- Selenium WebDriver is sufficient to write and run tests on a single machine, but need for a Selenium server(a more elaborate setup) is required if the tests need to run on browsers across different machines.
- More specifically, you will need Selenium Grid that allows the execution of automations in remote machines.
- Selenium Standalone server is a java jar file used to start the Selenium server. It is a smart proxy server that allows Selenium tests to route commands to remote web browser instances. The aim is to provide an easy way to run tests in parallel on multiple machines.
- To use a Selenium Grid, one can download the selenium-server-standalone JAR file and run it in standalone mode. You simply execute the .jar file, passing the mode as standalone to start the Selenium Standalone server.

```
java -jar selenium-server-4.3.0.jar standalone
```

WebDriver Commands

- Selenium WebDriver commands refer to the predefined methods (such as getTitle(), getCurrentUrl(), click(), and more) provided by Selenium WebDriver that enables interaction with web elements and control browser behavior during test automation.
- We need to learn the various commands that will be used throughout Selenium test scripts to automate testing of applications.
- The commands or '**Selenese**' can be divided into three categories:
  - Browser Commands
  - Navigation Commands
  - WebElement Commands

Browser Commands

The very first thing you like to do with Selenium is to:

- Open a new browser
- Perform few tasks
- Close the browser
- Below are the commands you can apply on the Selenium opened browser

Command	Java
Open a browser to a webpage	driver.get()
Get the title of the page you're on	driver.getTitle()
Close the current active browser tab	driver.close()
Close all the selenium managed browser tabs	driver.quit()

Syntax for Browser Commands

driver.get()

- This method **loads** a new web page in the current browser window.
- Accepts String as a parameter and returns nothing.
- Syntax - *driver.get(URL)*

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

// OR

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

driver.getTitle()

- This fetches the Title of the current page.

```
driver.getTitle();

// OR

String pageTitle = driver.getTitle();
```

Syntax for Browser Command

driver.close() / driver.quit():

- *close()* closes only the **current** window the *WebDriver* is currently controlling.
- *quit()* closes **all** the windows opened by the *WebDriver*.
- Both accept nothing as parameters and return nothing.
- Syntax - *driver.close()* OR *driver.quit()*

```
driver.close();

// OR

driver.quit();
```

Navigation Commands

- Navigation commands expose the ability to move backwards and forwards in the browser's history.
- **In Java,** you can access the navigation method through **driver.navigate().**

Command	**Java**
Navigate to another page	driver.navigate().to()
Go back in your history	driver.navigate().back()
Go forwards in your history	driver.navigate().forward()
Refresh the page	driver.navigate().refresh()

Syntax of Navigation Commands

navigate().to():

- This method loads a new web page in the current browser window.
- It accepts a String parameter and returns nothing.

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

navigate().forward():

- This method does the same operation as clicking on the Forward Button of any browser.

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

Syntax of Navigation Commands

navigate().back():

- This method does the same operation as clicking on the **Back Button** of any browser.

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

navigate().refresh():

- This method refreshes the current page.

```
driver.navigate().refresh();
```

Anatomy of a Selenium Script

- **Below is an example Selenium script.**

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

public class ClassName {
  public static void main(String[] args) {
    // Create a new instance of the Firefox driver
    WebDriver driver = new FirefoxDriver();

    // Open the browser
    driver.get("https://training-support.net");

    // Perform testing and assertions
    ...

    // Close the browser
    // Feel free to comment out the line below
    // so it doesn't close too quickly
    driver.quit();
  }
}
```

Anatomy of a Selenium Script

- The **WebDriver** and the **FirefoxDriver** classes are used to create an instance of the WebDriver class for Mozilla Firefox.
- All code should be in the *main()* method
- The *get()* method is used to open a Firefox window with the URL specified.
- *close()* is used to close the browser after test operations have been completed.

Basic Components of Selenium Script:

1. **Start the session:** WebDriver driver = new ChromeDriver();

2. **Take action on browser:** driver.get("https://www.selenium.dev/selenium/web/web-form.html");

3. **Request browser information:** driver.getTitle();

4. **Establish Waiting Strategy:** driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));

Anatomy of a Selenium Script

5. **Find an element:**

```
WebElement textBox = driver.findElement(By.name("my-text"));
WebElement submitButton = driver.findElement(By.cssSelector("button"));
```

6. Take action on element

```
textBox.sendKeys("Selenium");
submitButton.click();
```

7. Request element information

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

8. End the session: driver.quit();

Activities

Activity 1

Activity 1

Open a website

Using Selenium:

- Open a new browser to https://www.google.com/
- Get the title of the page and print it to the console.
- Write an Assertion to verify the page title
- Close the browser

Activity1

```
// Imports
import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class Activity1 {
    public static void main(String[] args) {
        // Set up Firefox driver
        WebDriverManager.firefoxdriver().setup();
        // Create a new instance of the Firefox driver
        WebDriver driver = new FirefoxDriver();
       //Open the page
        driver.get("https://www.google.com/");
        System.out.println("Home page title: " + driver.getTitle());
        //Assertion
        assertEquals(driver.getTitle(), "Google");
       //Close the browser
        driver.quit();
       }
  }
```

Locator Strategies

Locator Strategies

- Locators are the fundamental mechanism Selenium WebDriver uses to find and interact with specific elements (like buttons, text fields, links, images, etc.) on a web page.
- Just as you identify people by name or ID, Selenium identifies web elements using unique (or sufficiently specific) attributes.
- Without reliable locators, your automation scripts cannot accurately target elements, leading to "element not found" errors and unreliable tests.

Locator Strategies

- When Selenium successfully identifies an element on a web page using a locator, it returns a `WebElement` object (or a list of `WebElements` if multiple matches are found).
- This `WebElement` object represents the actual HTML element in the browser. You can then perform actions on this object (e.g., `click()`, `sendKeys()`, `submit()`) or retrieve information from it (e.g., `getText()`, `getAttribute()`, `isDisplayed()`).
- All interactions with the web page after finding an element are done through the `WebElement` object.

Locator Strategies

These are the primary methods for identifying `WebElement(s)` on a web page:

1. By ID (`By.id()`)

- Finds element by unique `id` attribute. (Most reliable)
- *Example:* `driver.findElement(By.id("username"));`

2. By Name (`By.name()`)

- Finds element by `name` attribute.
- *Example:* `driver.findElement(By.name("password"));`

Locator Strategies

3. By Class Name (`By.className()`)

- Finds element by `class` attribute.
- *Example:*
  `driver.findElement(By.className("btn-primary"));`

4. By Tag Name (`By.tagName()`)

- Finds element by its HTML tag (e.g., `input`, `div`, `a`).
- *Example:* `driver.findElement(By.tagName("h1"));`

5. By Link Text (`By.linkText()`)

- Finds <a> tag by its *exact* visible text.
- *Example:* `driver.findElement(By.linkText("Go to Homepage"));`

Locator Strategies

6. By Partial Link Text (`By.partialLinkText()`)

- Finds <a> tag by *partial* visible text.
- *Example:*
  ```
  driver.findElement(By.partialLinkText("Homepage"));
  ```

7. By CSS Selector (`By.cssSelector()`)

- Finds element using CSS selector syntax. (Powerful, often preferred over XPath)
- *Example:*
  ```
  driver.findElement(By.cssSelector("input[type='submit'][value='Login']"));
  ```

Locator Strategies

8. By XPath (`By.xpath()`)

- Finds element using XML Path Language expressions. (Most flexible, can navigate complex DOM)
- *Example:*
  ```
  driver.findElement(By.xpath("//button[text()='Submit']
  "));
  ```

Locator Strategies

By ID

- Locates an element by its unique `id` attribute.
- Highly reliable and preferred because `id` attributes are *supposed to be unique*

java

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

// Assuming 'driver' is an initialized WebDriver instance
// WebDriver driver = new ChromeDriver();
WebElement usernameField = driver.findElement(By.id("username"));
usernameField.sendKeys("myuser");
```

HTML

```html
<input type="text" id="username" name="user">
```

Introduction to XPath

What is XPath?

- XPath stands for XML Path Language
- XPath uses "path like" syntax to identify and navigate nodes in an XML document
- XPath contains over 200 built-in functions

XPath uses path expressions to select nodes or node-sets in an XML document.

There are two types of XPath:

- **Absolute XPath**
- **Relative XPath**

Types of XPath

Absolute XPath:

- It is a direct way to find the element.
- The disadvantage of absolute XPath is that if there are any changes made in the structure of the DOM, the XPath expression has to be rewritten.
- The key characteristic of XPath is that it begins with the single forward slash(/), which means you can select the element from the root node.

Absolute Xpath

```
<!-- Absolute XPath -->
/html/body/div[1]/section/div[1]/div/div/div/div[1]/div/div/div/div/div[3]
```

Types of XPath

Relative xpath:

- In Relative Xpath, the path starts from the middle of the HTML DOM structure.
- It starts with the double forward slash (//), which means it can search the element anywhere at the webpage.
- You can start from the middle of the HTML DOM structure and no need to write long absolute xpath.

Relative Xpath

```
<!-- Relative XPath -->
//div[@class='featured-box']
```

XPath Nodes

In XPath, there are seven kinds of nodes:

- element
- attribute
- text
- namespace
- processing-instruction
- comment
- document nodes.

XPath Terminology

...

```
<bookstore>
    <book>
        <title lang="en">Harry Potter and the Philosopher's Stone</title>
        <author>J K. Rowling</author>
        <year>1997</year>
        <price>29.99</price>
    </book>
    <book>
        <title lang="en">Percy Jackson and the Lightning Thief</title>
        <author>Rick Riordan</author>
        <year>2005</year>
        <price>39.99</price>
    </book>
    <book>
        <title lang="en">Leviathan Wakes</title>
        <author>James Corey</author>
        <year>2011</year>
        <price>39.99</price>
    </book>
</bookstore>
```

XPath Nodes

- **Parent Node**: Each element and attribute has one parent. The <book> element is the parent of the title, author, year, and price.
- **Child Nodes**: Element nodes may have zero, one or more children. The <title> <author>, <year>, and <price> elements are all children of the <book> element.
- **Sibling Nodes**: Nodes that have the same parent. The <title>, <author>, <year>, and <price> elements are all siblings.
- **Ancestor Nodes**: A node's parent, parent's parent, etc. The ancestors of the <title> element are the <book> element and the <bookstore> element.
- **Descendant Nodes**: A node's children, children's children, etc. Descendants of the <bookstore> element are the <book>, <title>, <author>, <year>, and <price> elements.

XPath Expressions

XPath uses path expressions to select nodes in an XML document. The node is selected by following a path or steps.

Xpath Expression	Meaning
bookstore	Select all nodes with the name “bookstore”
/bookstore	Selects a bookstore element which is a root element.
bookstore/book	Selects all book elements that are children of bookstore.
//book	Selects all book elements no matter where they are.
bookstore//book	Selects all book elements that are descendants of bookstore.

XPath Expressions

Xpath Expression	Meaning
//@lang	Selects all elements that have an attribute named lang.
/bookstore/book[1]	Selects the first book element that is a child of the bookstore.
/bookstore/book[last()]	Selects the last book element that is a child of the bookstore.
/bookstore/book[last()-1]	Selects the last but one book element that is a child of bookstore.
/bookstore/book[position()<3]	Selects the first 2 book elements that are children of bookstore.

XPath Expressions

Xpath Expression	Meaning
//title[@lang]	Selects all the title elements that have a lang attribute.
//title[@lang="en"]	Selects all the title elements that have a lang attribute that is "en".
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00.
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00.

XPath Methods

Finding elements with innerHTML text:

- By using *contains()* method in XPath, we can extract all the elements which matches a particular text value.
- The *contains()* method has an ability to find the element with partial text.
- Example:

```
...

<!-- To find a link with the text "About Us" -->
//a[contains(text(), 'About Us')]

<!-- Can be used with attributes as well -->
//button[contains(@class, 'green')]
//a[contains(@class, 'button')]
```

WebElement Commands

WebElement Commands

- These commands help in identifying the different Web Elements on webpages and performing various actions on it.
- A **WebElement** is an HTML Element.
- Some of the most commonly used WebElement commands are:

Description	**Java**
Click on an element	element.click()
Type into an element	element.sendKeys()
Get the text inside an element	element.getText()
Get the value of an attribute	element.getAttribute()
Get the value of a CSS property	element.getCssValue()

WebElement Commands

Clicking on Elements:

- Clicking is one of the most common way of interacting with web elements like text elements, links, radio boxes and many more**.**

```
WebElement element = driver.findElement(By.linkText("click here"));
element.click();
// OR
driver.findElement(By.linkText("click here")).click();
```

Typing into Elements:

- This simulates typing into an input element, like text boxes.

```
WebElement element = driver.findElement(By.id("username"));
element.sendKeys("username");
// OR
driver.findElement(By.id("username")).sendKeys("username");
```

WebElement Commands

Getting an Element's text: This returns the text of an element.

```
WebElement element = driver.findElement(By.linkText("anyLink"));
String linkText = element.getText();
```

Getting the value of an attribute:

This method gets the value of the given attribute of the element.

```
WebElement element = driver.findElement(By.id("SubmitButton"));
String attValue = element.getAttribute("id");
```

WebElement Commands

Get the value of a CSS Property:

- This method gets the value of the given CSS property of the element.

```
WebElement element = driver.findElement(By.id("SubmitButton"));
String cssValue = element.getCssValue("color");
```

Basic Actions on Web Elements

Basic Actions

What are Basic Actions?

- After successfully identifying a `WebElement` using a locator, you need to interact with it.
- "Basic Actions" are the fundamental methods provided by the `WebElement` interface to simulate user interactions and retrieve information from the web page.
- These actions form the building blocks of all automation scripts.

Basic Actions

Some of basic actions:

- **`click()`**: Simulates a mouse click on an element (e.g., button, link, checkbox).
- **`sendKeys()`**: Types text or sends special key presses (like Enter) into an input field.
- **`clear()`**: Clears the existing text content from an editable input field.
- **`submit()`**: Submits the form associated with the element it's called on.
- **`getText()`**: Retrieves the visible, rendered text content of an element.
- **`getAttribute("name")`**: Gets the value of a specified HTML attribute (e.g., `href`, `value`).

Basic Actions

`isDisplayed()`: Checks if an element is visible on the page (returns `boolean`).
`isEnabled()`: Checks if an element is enabled for interaction (returns `boolean`).
`isSelected()`: Checks if a checkbox, radio button, or option is selected (returns `boolean`).
`getTagName()`: Returns the HTML tag name of the element (e.g., "div", "input").
`getCssValue("property")`: Retrieves the computed value of a CSS property (e.g., "font-size", "color").

Basic Actions

1. Clicking (`click()`)

- Simulates a mouse click on an interactive element (e.g., buttons, links, checkboxes, radio buttons, images).
- returns nothing (`void`)

click()

```
WebElement submitButton = driver.findElement(By.id("submitBtn"));
submitButton.click(); //Clicks the button
```

Basic Actions

Entering Text (`sendKeys()`)

- Types a character sequence into an editable text field (e.g., `input` text, `textarea`).
- Can also send special keys: Like `Keys.ENTER`, `Keys.TAB`, `Keys.ARROW_DOWN`, etc.
- Return Type: `void`

enter text

```
WebElement searchBox = driver.findElement(By.name("q"));
searchBox.sendKeys("Selenium WebDriver tutorial"); // Types text
searchBox.sendKeys(Keys.ENTER); // Presses Enter key
```

Basic Actions

Submitting a Form (`submit()`):

- Submits the form that contains the `WebElement` this method is called on (e.g., an input field or a button within a form).

submit

```
WebElement usernameInput = driver.findElement(By.id("username"));
usernameInput.sendKeys("myuser");
usernameInput.submit(); // Submits the enclosing form
```

Basic Actions

Getting Element Text (`getText()`):

- Retrieves the visible (rendered) inner text of the `WebElement`, including text from any sub-elements. Returns a `String`.

getText()

```
WebElement pageTitleElement = driver.findElement(By.tagName("h1"));
String pageTitle = pageTitleElement.getText(); // Gets the text "Welcome!" from <h1>Welcome!
</h1>
System.out.println("Page Title: " + pageTitle);
```

Activity 2

Activity 2

Sending Input

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/login-form
- Get the title of the page and print it to the console.
- Find the username field **using any locator** and enter "admin" into it.
- Find the password field **using any locator** and enter "password" into it.
- Find the "Log in" button **using any locator** and click it.
- Close the browser.

Activity 3

Activity 3
Target Practice

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/target-practice
- Get the title of the page and print it to the console.
- **Using xpath**:
  - Find the 3rd header on the page and print it's text to the console.
  - Find the 5th header on the page and print it's color.
- **Using any other locator:**
  - Find the violet button and print all it's classes.
  - Find the grey button and print it's text.
- Close the browser.

Advanced Actions

Advanced Actions

- While `WebElement` methods like `click()` and `sendKeys()` handle most common interactions, some user scenarios require more sophisticated control.
- These scenarios involve combinations of mouse movements, key presses, and interactions that simulate how a real user would typically interact with dynamic or complex UI elements.

Advanced Actions

Introducing the `Actions` Class

- `Actions` class in Selenium WebDriver is specifically designed to handle advanced user interactions.
- Actions from from the package `org.openqa.selenium.interactions.Actions`
- It allows you to build a *sequence* of individual actions into a single, composite action.
- The `Actions` class follows the Builder pattern.
- You create an `Actions` object and then chain multiple action methods together before executing them.

Advanced Actions

How the `Actions` Class Works:

- Create an instance of the `Actions` class, passing your `WebDriver` instance to its constructor.
- `Actions actions = new Actions(driver);`
- Call various `Actions` methods to define the sequence of interactions. These methods return the `Actions` object itself, allowing for chaining.
- Call `.build()` to compile the chained actions into a single, ready-to-be-performed action sequence.
- Call `.perform()` to execute the entire composite action sequence in the browser.

Advanced Actions

Common Advanced Actions

1. Mouse Actions:

- **`contextClick(element)`**: Performs a right-click (context-click) on an element.
- **`doubleClick(element)`**: Performs a double-click on an element.
- **`moveToElement(element)`**: Moves the mouse cursor to the center of a specified element (for hovering).
- **`dragAndDrop(source, target)`**: Drags a source element and drops it onto a target element.
- **`dragAndDropBy(source, xOffset, yOffset)`**: Drags a source element by a given pixel offset.

Advanced Actions

Right-Click (Context Click) (`contextClick()`):

- Simulates a right-mouse click on a specific element or the current mouse position. This often brings up a context menu.

Advanced Actions

Keyboard Actions:

- **`keyDown(key)`**: Presses and holds down a modifier key (e.g., `CONTROL`, `SHIFT`, `ALT`).
- **`keyUp(key)`**: Releases a pressed modifier key.
- **`sendKeys(keysToSend)`**: Sends a sequence of keys to the currently focused element or globally.

Activity 4

Activity 4

Input Events #1

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/input-events
- Get the title of the page and print it to the console.
- On the page, perform:
  - Left click and print the value of the side in the front.
  - Double click to show a random side and print the number.
  - Right click and print the value shown on the front of the cube.
- Close the browser.

Activity 5

Activity 5

Input Events #2

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/input-events
- Get the title of the page and print it to the console.
- On the page, perform:
    - Press the key of first letter of your name in caps
    - Press CTRL+a and the CTRL+c to copy all the text on the page.
- Close the browser.

Activity 6

Activity 6
Drag and Drop

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/drag-drop
- Get the title of the page and print it to the console.
- On the page, perform:
    - Find the ball and simulate a click and drag to move it into "Dropzone 1".
    - Verify that the ball has entered Dropzone 1.
    - Once verified, move the ball into "Dropzone 2".
    - Verify that the ball has entered Dropzone 2.
- Close the browser.

Selenium Waits

Application Synchronization

- Application synchronization in Selenium refers to the techniques used to coordinate your test execution with the dynamic behavior of web applications.
- This is crucial because web pages load asynchronously, and elements may appear, disappear, or change state at different times. Without proper synchronization, tests become flaky and unreliable, leading to false failures.
- WebDriver can generally be said to have a blocking API. Since it is an out-of-process library that instructs the browser what to do, and because the web platform has an intrinsically asynchronous nature, WebDriver does not track the active, real-time state of the DOM.
- Most intermittent issues that arise from use of Selenium and WebDriver are connected to ***race conditions*** that occur between the browser and the user's instructions. **An example could be that the user instructs the browser to navigate to a page, then gets a "*no such element*" error when trying to find an element.**

Application Synchronization

- The issue here is that the default page load strategy used in WebDriver listens for the **document.readyState** to change to "**complete**" before returning from the call to ***navigate***.
- If an element is added *after* the document has completed loading, the WebDriver script *might* not catch that change.
- The first solution to this is adding a sleep statement to pause the code execution for a set period of time. Because the code can't know exactly how long it needs to wait, this can fail when it doesn't sleep long enough.
- Alternately, if the value is set too high and a sleep statement is added in every place it is needed, the duration of the session can become prohibitive.
- Selenium provides two different mechanisms for synchronization that are better:

1. Implicit Wait
2. Explicit Wait

Implicit Wait

- Selenium has a built-in way to automatically wait for elements called an *implicit wait*. An implicit wait value can be set either with the timeouts capability in the browser options, or with a driver method.
- But, **the default setting is 0, meaning disabled. Once manually enabled, the implicit wait is set for the life of the session.** Implicit wait needs to be set on a per-session basis.
- This is a global setting that applies to every element location call for the entire session. The default value is `0`, which means that if the element is not found, it will immediately return an error.
- If an implicit wait is set, the driver will wait for the duration of the provided value before returning the error. Note that as soon as the element is located, the driver will return the element reference and the code will continue executing, so a larger implicit wait value won't necessarily increase the duration of the session.
- An implicit wait is to tell WebDriver to poll the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available.

Using Implicit Waits

- A WebDriver `session` is imposed with a certain `session timeout` interval, during which the user can control the behaviour of executing scripts or retrieving information from the browser.
- In the below example, we are setting an implicit wait of 20 seconds.

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(20));
```

- Implicit wait makes WebDriver to wait for a specified amount of time when trying to locate an element before throwing a NoSuchElementException.

Explicit Waits

- **Explicit wait** in Selenium is a synchronization mechanism that allows the WebDriver to wait for a specific condition to occur before proceeding with the next step in the code.
- Since explicit waits allow you to wait for a condition to occur, they make a good fit for synchronising the state between the browser and its DOM, and your WebDriver script.
- Unlike Implicit waits, which apply globally, explicit waits are applied only to specific elements or conditions, making them more flexible, intelligent and precise.It is an improvement on implicit wait since it allows the program to pause for dynamically loaded Ajax elements. But explicit waits can be applied only for specified elements.
- The condition is called with a certain frequency until the timeout of the wait is elapsed.
- **This means that for as long as the condition returns a false value, it will keep trying and waiting.**

Explicit Waits

- Setting Explicit Wait is important in cases where there are certain elements that naturally take more time to load. If one sets an implicit wait command, then the browser will wait for the same time frame before loading every web element. This causes an unnecessary delay in executing the test script.
- You can use explicit waits in one of two ways:
  - WebDriverWait
  - FluentWait
- To use Explicit Wait in test scripts, import the following packages into the script.

```
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
```

Explicit Wait - WebDriver Wait

- **WebDriverWait** is applied on certain element with defined *expected condition* and *time*. This wait is only applied to the specified element.
- This wait throws exception when element is not found.
- It is commonly used with the **ExpectedConditions** class.

```
WebDriverWait wait = new WebDriverWait(WebDriverRefrence, Duration);

// Example
WebDriverWait wait = new WebDriverWait (driver, Duration.ofSeconds(20));
wait.until(ExpectedConditions.VisibilityofElementLocated(By.id("heading")));
```

Expected Conditions

- Because it is quite a common occurrence to have to synchronise the DOM and your instructions, most clients also come with a set of predefined *expected conditions*.
- As might be obvious by the name, they are conditions that are predefined for frequent wait operations.
- Expected Conditions are set as below:

```
wait.until(ExpectedConditions.VisibilityofElementLocated(By.id("heading")));
```

- You can get a full list of available Expected conditions by clicking here.

Few best practices for using Expected conditions:

- It is important to choose appropriate expected condition for the scenario under test.
- Keep Waits Close to Action: Place waits just before interacting with dynamic elements to improve script reliability.

Common Wait conditions

- Some common wait conditions you will be using are:

You're waiting for	Java
An Alert to be present	alertIsPresent()
An element to be clickable	elementToBeClickable()
The title of the page to be a certain string	titleIt(String)
The title of the page to contain a certain string	titleContains(String)
An element to become visible	visibilityOfElementLocated(By)

Explicit Wait - Fluent Wait

- **FluentWait** is like **WebDriverWait**, but it allows you to configure additional things about the wait itself, such as:
  - How often it should check for the condition to be satisfied?
  - Which exceptions should it ignore?
- This command operates with two primary parameters: timeout value and polling frequency.

```
// Waiting 30 seconds for an element to be present on the page, checking
// for its presence once every 5 seconds.
Wait wait = new FluentWait(driver)
  .withTimeout(Duration.ofSeconds(30))
  .pollingEvery(Duration.ofSeconds(5))
  .ignoring(NoSuchElementException.class);

WebElement foo = wait.until(driver -> {
  return driver.findElement(By.id("foo"));
});
```

Implicit vs Explicit Wait

Implicit Wait in Selenium	Explicit Wait in Selenium
Applies to all elements in a test script.	Applies only to specific elements as intended by the user.
No need to specify "ExpectedConditions" on the element to be located	Must always specify "ExpectedConditions" on the element to be located
Most effective when used in a test case in which the elements are located with the time frame specified in implicit wait	Most effective when used when the elements are taking a long time to load. Also useful for verifying property of the element, such as visibilityOfElementLocated, elementToBeClickable, elementToBeSelected

Activity 7

Activity 7

Waits #1

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/dynamic-controls
- Get the title of the page and print it to the console.
- On the page, perform:
    - Find the checkbox toggle button and click it.
    - Wait till the checkbox disappears.
    - Click the button again. Wait till it appears and check the checkbox.
- Close the browser.

Activity7

```
// Set up Firefox driver
WebDriverManager.firefoxdriver().setup();
// Create a new instance of the Firefox driver
WebDriver driver = new FirefoxDriver();
// Create the Wait object
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

// Open the page
driver.get("https://v1.training-support.net/selenium/dynamic-controls");
// Print the title of the page
System.out.println("Home page title: " + driver.getTitle());

// Find the toggle button and click it
WebElement toggleButton = driver.findElement(By.id("toggleCheckbox"));
toggleButton.click();
// Wait for the toggleButton to disappear
WebElement dynamicBox = driver.findElement(By.id("dynamicCheckbox"));
wait.until(ExpectedConditions.invisibilityOf(dynamicBox));
System.out.println(dynamicBox.isDisplayed());
// Click the button again
toggleButton.click();
// Wait for the element to appear
wait.until(ExpectedConditions.visibilityOf(dynamicBox));
System.out.println(dynamicBox.isDisplayed());

// Close the browser
driver.close();
```

Activity 8

Activity 8

Waits #2

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/ajax
- Get the title of the page and print it to the console.
- On the page, perform:
    - Find and click the "Change content" button on the page.
    - Wait for the text to say "HELLO!". Print the message that appears on the page.
    - Wait for the text to change to contain "I'm late!". Print the new message on the page.
- Close the browser.

Activity8

```
      // Setup the driver
        WebDriverManager.firefoxdriver().setup();
        // Driver object reference
        WebDriver driver = new FirefoxDriver();
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

        // Open the browser
        driver.get("https://v1.training-support.net/selenium/ajax");

        // Find the button and click it
        driver.findElement(By.cssSelector("button.violet")).click();
        // Wait for the new elements to appear
        wait.until(ExpectedConditions.visibilityOfElementLocated(By.tagName("h1")));
        // Find and print the new text
        String text = driver.findElement(By.tagName("h1")).getText();
        System.out.println(text);

        WebElement delayedText = driver.findElement(By.tagName("h3"));
        System.out.println(delayedText.getText());
        // Wait for the delayed text and print it
        wait.until(ExpectedConditions.textToBePresentInElementLocated(By.tagName("h3"), "I'm
late!"));
        String lateText = driver.findElement(By.tagName("h3")).getText();
        System.out.println(lateText);

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

Verifying WebElements

Verifying WebElements

To verify the properties of WebElements the methods used are:

Description	**Method**
Check if an element is visible on the page	element.isDisplayed()
Check if an input element can be interacted with	element.isEnabled()
Check if a checkbox or radio button is selected	element.isSelected()

Verifying WebElements

isDisplayed()

- This method is used to check whether an element is displayed on a web page or not.
- Returns true if the target element is displayed otherwise returns false.

isDisplayed()

```
driver.findElement(By.name("username")).isDisplayed();
```

Verifying WebElements

isEnabled()

- It is used to check if the web element is enabled or disabled within the web page.
- Returns “true” value if the specified web element is enabled on the web page
- Returns “false” value if the web element is disabled on the web page.

isEnabled()

```
driver.findElement(By.name("username")).isEnabled();
```

Verifying WebElements

isSelected()

- This method checks that if an element is selected on the web page or not.
- Returns true if the element is selected.
- Returns false if the element is not selected.
- It can be executed only on a radio button, checkboxes, etc.

isSelected()

```
driver.findElement(By.id("opt-in-checkbox")).isSelected();
```

Activity 9

Activity 9

Check if elements are displayed

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/dynamic-controls
- Get the title of the page and print it to the console.
- On the page, perform:
  - Find the checkbox input element.
  - Check if it is visible on the page.
  - Click the "Remove Checkbox" button.
  - Check if it is visible again and print the result.
- Close the browser.

Activity9

```
 // Set up Firefox driver
        WebDriverManager.firefoxdriver().setup();
        // Create a new instance of the Firefox driver
        WebDriver driver = new FirefoxDriver();
        // Create the Wait object
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

        // Open the page
        driver.get("https://v1.training-support.net/selenium/dynamic-controls");
        // Print the title of the page
        System.out.println("Home page title: " + driver.getTitle());

        // Find the checkbox
        WebElement checkbox = driver.findElement(By.className("willDisappear"));
        // Find the toggle button and click it
        WebElement checkboxToggle = driver.findElement(By.id("toggleCheckbox"));
        checkboxToggle.click();
        // Wait for the checkbox to disappear

wait.until(ExpectedConditions.invisibilityOfElementLocated(By.className("willDisappear")));
        System.out.println("Checkbox is displayed: " + checkbox.isDisplayed());
        // Click the button again
        checkboxToggle.click();
        // Wait for the element to appear
        wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("dynamicCheckbox")));
        System.out.println("Checkbox is displayed: " + checkbox.isDisplayed());
        // Click the checkbox
        driver.findElement(By.xpath("//input[@class='willDisappear']")).click();

        // Close the browser
        driver.close();
```

Activity 10

Activity 10

Check if elements are selected

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/dynamic-controls
- Get the title of the page and print it to the console.
- On the page, perform:
    - Find the checkbox input element.
    - Check if the checkbox is selected and print the result.
    - Click the checkbox to select it.
    - Check if the checkbox is selected again and print the result.
- Close the browser.

Activity10

```
// Set up Firefox driver
  WebDriverManager.firefoxdriver().setup();
  // Create a new instance of the Firefox driver
  WebDriver driver = new FirefoxDriver();
  // Create the Wait object
  WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

  // Open the page
  driver.get("https://v1.training-support.net/selenium/dynamic-controls");
  // Print the title of the page
  System.out.println("Home page title: " + driver.getTitle());

  // Find the checkbox and click it
  WebElement checkbox = driver.findElement(By.name("toggled"));
  checkbox.click();
  System.out.println("Checkbox is selected: " + checkbox.isSelected());
  // Click the checkbox to deselect it
  checkbox.click();
  System.out.println("Checkbox is selected: " + checkbox.isSelected());

  // Close the browser
  driver.close();
```

Activity 11

Activity 11

Check if elements are enabled

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/dynamic-controls
- Get the title of the page and print it to the console.
- On the page, perform:
    - Find the text field.
    - Check if the text field is enabled and print it.
    - Click the "Enable Input" button to enable the input field.
    - Check if the text field is enabled again and print it.
- Close the browser.

Activity11

```
// Set up Firefox driver
        WebDriverManager.firefoxdriver().setup();
        // Create a new instance of the Firefox driver
        WebDriver driver = new FirefoxDriver();
        // Create the Wait object
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

        // Open the page
        driver.get("https://v1.training-support.net/selenium/dynamic-controls");
        // Print the title of the page
        System.out.println("Home page title: " + driver.getTitle());

        // Find the text field
        WebElement textbox = driver.findElement(By.id("input-text"));
        // Check if it is enabled
        System.out.println("Input field is enabled: " + textbox.isEnabled());
        // Click the toggle button to enable it
        driver.findElement(By.id("toggleInput")).click();
        // Check if the text field is enabled
        System.out.println("Input field is enabled: " + textbox.isEnabled());

        // Close the browser
        driver.close();
```

Handling Dynamic Attributes

Handling Dynamic Attributes

- Dynamic elements are database-driven.
- When you update one element in a database, it affects a number of areas on the website application.
- Dynamic identifiers are normally used for **buttons**, **text-fields** and **form elements**.
- Tests for dynamic sites have to be rewritten *each time the page updates*, because the identifiers are different each time the page loads.
- For example, an attribute '**bttn-3455-textE1**' will change to '**bttn-3456-textE1**' next time the page loads.

Handling Dynamic Attributes

Locate Elements by Starting, Ending or Containing Text

If the elements have ID or css-attribute **values with static text in start, end or in between**, then we can write **XPath** or **CSS selectors** based on this.

Let’s take a look into it one by one.

Handling Dynamic Attributes

If element has attribute value where **starting text** remains constant, then we can utilize **CSS** (*[attribute^=value]*) and **XPath** (*starts-with*) functions to create XPath or CSS selectors like below:

```
...

/* Example HTML */
/* On 1st page load */
<button id="downshift-main-jss373"></button>
/* On page reload */
<button id="downshift-main-jss981"></button>

/* CSS */
button[id^='downshift-main-']

/* XPath */
//button[starts-with(@id, 'downshift-main-')]
```

Handling Dynamic Attributes

Similarly, if element attribute value has static **ending text**, then we can create selectors by using **CSS** (*[attribute$=value]*) or **XPath** (*ends-with*) as shown below:

```
...

/* Example HTML */
/* On 1st page load */
<button id="jss903-downshift-main"></button>
/* On page reload */
<button id="jss981-downshift-main"></button>

/* CSS */
button[id$='-downshift-main']

/* XPath */
// No XPath variant available in modern browsers
```

Handling Dynamic Attributes

Finally, if the ending or starting text are not static, but there is some **text in between** that is static, then we can use **CSS** (*[attribute*=value]*) and **XPath** (*contains*) functions to create XPath or CSS selectors like below:

```
...

/* Example HTML */
/* On 1st page load */
<li role="jss21-user-option-901"></li>
/* On page reload */
<li role="jss53-user-option-873"></li>

/* CSS */
li[role*='-user-option-']

/* XPath */
//li[contains(@role, '-user-option-')]
```

Activity 12

Activity 12

Dynamic Attributes #1

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/dynamic-attributes
- Get the title of the page and print it to the console.
- Find the username and password input fields. Type in the credentials:
  - username: *admin*
  - password: *password*
- Wait for login message to appear and print the login message to the console.
- Close the browser.

Activity12

```
      // Set up Firefox driver
        WebDriverManager.firefoxdriver().setup();
        // Create a new instance of the Firefox driver
        WebDriver driver = new FirefoxDriver();
        // Create the Wait object
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

        // Open the page
        driver.get("https://v1.training-support.net/selenium/dynamic-attributes");
        // Print the title of the page
        System.out.println("Home page title: " + driver.getTitle());

        // Find the username and password fields
        WebElement username = driver.findElement(By.xpath("//input[starts-with(@class,
'username-')]"));
        WebElement password = driver.findElement(By.xpath("//input[starts-with(@class,
'password-')]"));
        // Enter the credentials
        username.sendKeys("admin");
        password.sendKeys("password");
        // Find and click the submit button
        driver.findElement(By.xpath("//button[@type='submit']")).click();

        // Print the login message
        String message = driver.findElement(By.id("action-confirmation")).getText();
        System.out.println("Login message: " + message);

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

Activity 13

Activity 13

Dynamic Attributes #2

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/dynamic-attributes
- Get the title of the page and print it to the console.
- Find the input fields of the Sign Up form.
- Fill in the details in the fields with your own data.
- Wait for success message to appear and print it to the console.
- Close the browser.

Activity13

```
// Set up Firefox driver
        WebDriverManager.firefoxdriver().setup();
        // Create a new instance of the Firefox driver
        WebDriver driver = new FirefoxDriver();
        // Create the Wait object
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

        // Open the page
        driver.get("https://v1.training-support.net/selenium/dynamic-attributes");
        // Print the title of the page
        System.out.println("Home page title: " + driver.getTitle());

        // Find the fields of the sign-up form
        WebElement userName = driver.findElement(By.xpath("//input[contains(@class, '-
username')]"));
        WebElement password = driver.findElement(By.xpath("//input[contains(@class, '-
password')]"));
        WebElement confirmPassword = driver.findElement(By.xpath("//label[text() = 'Confirm
Password']/following-sibling::input"));
        WebElement email = driver.findElement(By.xpath("//label[contains(text(),
'mail')]/following-sibling::input"));

        // Type credentials
        userName.sendKeys("NewUser");
        password.sendKeys("Password");
        confirmPassword.sendKeys("Password");
        email.sendKeys("real_email@xyz.com");
        // Click Sign Up
        driver.findElement(By.xpath("//button[contains(text(), 'Sign Up')]")).click();

        // Print login message
        String loginMessage = driver.findElement(By.id("action-confirmation")).getText();
        System.out.println("Login message: " + loginMessage);

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

Handling Selects

Handling Selects

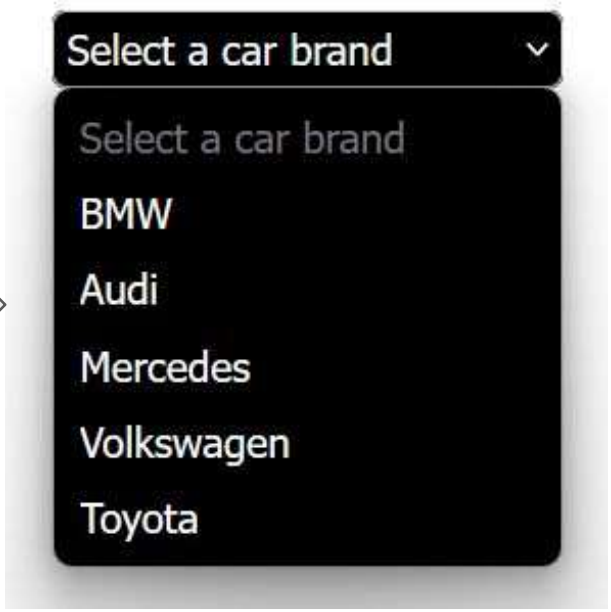
The HTML <select> element represents a control that provides a menu of options:

- The **<select>** element is typically used to create a drop-down list.
- The **<option>** tags inside the **<select>** element define the available options in the list.
- Adding a **'multiple'** attribute to a **<select>** tag allows us to select multiple options.

Handling Selects

HTML

```
<select>
  <option value="" disabled selected>Select a car brand</option>
  <option value="bmw">BMW</option>
  <option value="audi">Audi</option>
  <option value="mercedes">Mercedes</option>
  <option value="volkswagen">Volkswagen</option>
  <option value="toyota">Toyota</option>
</select>
```



Handling Selects

Multi Select

HTML

```
<select  multiple>
  <option value="java">Java</option>
  <option value="python">Python</option>
  <option value="javascript">JavaScript</option>
  <option value="csharp">C#</option>
  <option value="cpp">C++</option>
</select>
```

Handling Selects

Common Methods to Both Single & Multi-Select Dropdowns:

- **`new Select(WebElement)`**: Creates the helper to interact with a dropdown element.
- **`selectByVisibleText("Text")`**: Selects an option by its displayed text.
- **`selectByValue("Value")`**: Selects an option by its hidden 'value' attribute.
- **`selectByIndex(index)`**: Selects an option by its position (starting from 0).
- **`getFirstSelectedOption()`**: Gets the first currently selected item from the dropdown.
- **`getOptions()`**: Gets a list of all available options in the dropdown.
- **`isMultiple()`**: Checks if the dropdown allows selecting multiple items.

Handling Selects

Specific methods to Multi-Select Dropdowns (`<select multiple>`):

- **`deselectByVisibleText("Text")`**: Unselects an option by its displayed text.
- **`deselectByValue("Value")`**: Unselects an option by its hidden 'value' attribute.
- **`deselectByIndex(index)`**: Unselects an option by its position.
- **`deselectAll()`**: Unselects all currently selected options in a multi-select dropdown.
- **`getAllSelectedOptions()`**: Gets a list of all currently selected items from a multi-select dropdown.

Handling Selects

Select object initialization syntax:

Script

```
//Find element using locator
WebElement dropdown = driver.findElement(By.xpath("//select"));

//Initialize Select object
Select list = new Select(dropdown);
```

Handling Selects

Drop down

HTML

```html
<select>
    <option>Select an Option</option>
    <option value="one">Option 1</option>
    <option value="two">Option 2</option>
    <option value="three">Option 3</option>
    <option value="four">Option 4</option>
    <option value="five">Option 5</option>
    <option value="six">Option 6</option>
</select>
```

...

```java
// Store the select and make a new select object
WebElement dropdown = driver.findElement(By.tagName("select"));
Select select = new Select(dropdown);

// Get all the options inside the select as a list
List<WebElement> options = select.getOptions();

// Select the third option based on Index
select.selectByIndex(2)

// Selecting the fourth option based on it's value
select.selectByValue("four")

// Selecting the third option based on visible text
select.selectByVisibleText("Option 3")
```

Handling Selects

Multiple-select

```
...
```

```
<select id="bikeMultiSelect" multiple size="5">
        <option value="none">-- Select multiple --</option>
        <option value="hero">Hero Splendor</option>
        <option value="bajaj">Bajaj Pulsar</option>
        <option value="honda">Honda Activa</option>
        <option value="royal_enfield">Royal Enfield Classic</option>
        <option value="yamaha">Yamaha FZ</option>
    </select>
```

multiple select

```
// Select three bikes using different methods
multiBikeSelect.selectByIndex(1);          // Selects 'Hero Splendor' (index 1)
multiBikeSelect.selectByValue("honda");    // Selects 'Honda Activa' (value="honda")
multiBikeSelect.selectByVisibleText("Yamaha FZ"); // Selects 'Yamaha FZ'

// Deselect one of them using a different method
multiBikeSelect.deselectByValue("honda");  // Deselects 'Honda Activa'
```

Handling Selects

Get all Selected options

```
...
```

```
// Store the select and make a new select object
WebElement dropdown = driver.findElement(By.tagName("select"));
Select select = new Select(dropdown);

// Get all the options inside the select as a list
List<WebElement> options = select.getOptions();

/*
 * Getting ALL Selected Options
 */

List<WebElement> allSelected = select.getAllSelectedOptions();

/*
 * Getting The First Selected Option
 * According to the DOM
 */

WebElement firstOption = select.getFirstSelectedOption();
```

Activity 14

Activity 14

Selects #1

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/selects
- Get the title of the page and print it to the console.
- **On the Single Select**:
  - Select the second option using the visible text.
  - Select the third option using the index.
  - Select the fourth option using the value.
  - Get all the options and print them to the console.
- Close the browser.

Activity 14
Part - 1

Activity14 - 1

```
// Set up Firefox driver
        WebDriverManager.firefoxdriver().setup();
        // Create a new instance of the Firefox driver
        WebDriver driver = new FirefoxDriver();
        // Create the Wait object
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

        // Open the page
        driver.get("https://v1.training-support.net/selenium/selects");
        // Print the title of the page
        System.out.println("Home page title: " + driver.getTitle());

        // Find the dropdown
        WebElement dropdown = driver.findElement(By.id("single-select"));
```

Activity 14 Part - 2

Activity14 - 2

```
 // Pass the WebElement to the Select object
        Select singleSelect = new Select(dropdown);

        // Select the second option using visible text
        singleSelect.selectByVisibleText("Option 2");
        // Print the selected option
        System.out.println("Second option: " +
singleSelect.getFirstSelectedOption().getText());

        // Select the third option using index
        singleSelect.selectByIndex(3);
        // Print the selected option
        System.out.println("Third option: " +
singleSelect.getFirstSelectedOption().getText());

        // Select the fourth option using value attribute
        singleSelect.selectByValue("4");
        // Print the selected option
        System.out.println("Fourth option: " +
singleSelect.getFirstSelectedOption().getText());

        // Print all the options
        List<WebElement> allOptions = singleSelect.getOptions();
        System.out.println("Options in the dropdown: ");
        for(WebElement option : allOptions) {
            System.out.println(option.getText());
        }

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

Activity 15

Activity 15

Selects #2

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/selects
- Get the title of the page and print it to the console.
- **On the Multi Select**:
  - Select the "JavaScript" option using the visible text.
  - Select the 4th, 5th and 6th options using the index.
  - Select the "NodeJS" option using the value.
  - Deselect the 5th option using index.
- Close the browser.

Activity 15
Part - 1

Activity15 - 1

```
// Set up Firefox driver
        WebDriverManager.firefoxdriver().setup();
        // Create a new instance of the Firefox driver
        WebDriver driver = new FirefoxDriver();
        // Create the Wait object
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

        // Open the page
        driver.get("https://v1.training-support.net/selenium/selects");
        // Print the title of the page
        System.out.println("Home page title: " + driver.getTitle());

        // Find the dropdown
        WebElement selectElement = driver.findElement(By.id("multi-select"));
        // Pass the WebElement to the Select object
        Select multiSelect = new Select(selectElement);
```

Activity 15 Part - 2

Activity15 - 2

```
// Select "Javascript" using visible text
        multiSelect.selectByVisibleText("Javascript");
        // Select 4th, 5th, and 6th index options
        for(int i = 4; i<=6 ; i++) {
            multiSelect.selectByIndex(i);
        }
        // Select "NodeJS" using value attribute
        multiSelect.selectByValue("node");
        // Print the selected options
        List<WebElement> selectedOptions = multiSelect.getAllSelectedOptions();
        System.out.println("Selected options are: ");
        for(WebElement option : selectedOptions) {
            System.out.println(option.getText());
        }

        // Deselect the 5th index option
        multiSelect.deselectByIndex(5);
        // Print the selected options
        selectedOptions = multiSelect.getAllSelectedOptions();
        System.out.println("Selected options are: ");
        for(WebElement option : selectedOptions) {
            System.out.println(option.getText());
        }

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

Handling Alerts

Handling Alerts

- JavaScript alert in Selenium is a message/notification box that notifies the user about some information or asks for permission to perform a certain kind of operation.
- There are three types of Alert in Selenium, described as follows:
  - Simple Alert
  - Prompt Alert
  - Confirmation Alert
- Selenium's Java bindings come with an interface called "Alert" which makes handling Alerts very easy.
- The Alert interface is part of the org.openqa.selenium.Alert package

Handling Alerts

- The Alert interface gives us access to the following methods which help us handle alerts in a graceful manner:
  - accept() - To accept the alert. Same as clicking "OK"
  - dismiss() - To dismiss the alert. Same as clicking "Cancel"
  - getText() - To get the text of the alert
  - sendKeys() -To write some text to the alert
- To switch focus to the alert box, use driver.switchTo().alert()

Simple Alert

- The simplest of these is referred to as an alert, which shows a custom message, and a single button which dismisses the alert, labelled in most browsers as OK.
- It can also be dismissed in most browsers by pressing the close button, but this will always do the same thing as the OK button.

```
// Click the button that triggers the alert
driver.findElement(By.id("alert-button")).click();

// Wait for the alert to be displayed
wait.until(ExpectedConditions.alertIsPresent());

// Store the alert in a variable
Alert alert = driver.switchTo().alert();

// Print the text from the alert
System.out.println(alert.getText());

// Click the OK button on the alert
alert.accept();
```

Confirmation Alert

- This alert is basically used for the confirmation of some tasks. For Example: Do you wish to continue a particular task? Yes or No? The snapshot below depicts the same.
- A confirm box is similar to an alert, except the use can also choose to cancel the message by clicking the "cancel" button.

```
// Click the button that triggers the alert
driver.findElement(By.id("alert-button")).click();

// Wait for the alert to be displayed
wait.until(ExpectedConditions.alertIsPresent());

// Store the alert in a variable
Alert alert = driver.switchTo().alert();

// Print the text from the alert
System.out.println(alert.getText());

// Click the Cancel button on the alert
alert.dismiss();
```

Prompt Alerts

- Prompts are similar to confirm boxes, except they also include a text input.
- This alert will ask the user to input the required information to complete the task.

```
// Click the button that triggers the alert
driver.findElement(By.id("alert-button")).click();

// Wait for the alert to be displayed
wait.until(ExpectedConditions.alertIsPresent());

// Store the alert in a variable
Alert alert = driver.switchTo().alert();

// Print the text from the alert
System.out.println(alert.getText());

// Send text to the alert
alert.sendKeys("Hello!");

// Click the OK button on the alert
alert.accept();
```

Handling Multiple Windows/Tabs

Clicking a link which opens in a new window will focus the new window or tab on screen, but WebDriver will not know which window the operating system considers active.

- To work with the new window you will need to switch context to it.
- If you have only two tabs or windows open, and you know which window you start with, by the process of elimination you can loop over both windows or tabs that WebDriver can see, and switch to the one which is not the original.
- Each Tab/Window that is created by the WebDriver is assigned a unique handle.
- First, we use **driver.getWindowHandles()** to obtain a list containing the handles of all the currently open tabs/windows.
- Once we have the list, we iterate through the list until we get the handle of the tab that we require.
- Then, we use **driver.switchTo().window(<handle>)** to switch the driver's focus to that tab.

Handling Multiple Windows/Tabs

```java
// Open a Web Page
driver.get("https://training-support.net/webelements/tabs");

// Store the handle of the current tab in a variable
// This will help you find the newly opened tab later.
String parentHandle = driver.getWindowHandle();

// Click on the button that opens a new page in another tab
driver.findElement(By.id("launcher")).click();

// Wait until there's 2 tabs properly open
wait.until(ExpectedConditions.numberOfWindowsToBe(2));

// Store the handles of all the open tabs in a variable.
// Note: getWindowHandles() will always return a Set.
Set<String> handles = driver.getWindowHandles();

// Loop through the Set and switch to the window
// whose handle doesn't match the parentHandle
for(String handle : handles) {
    if (handle != parentHandle) {
        driver.switchTo().window(handle);
    }
}

// Wait for the page to actually load
wait.until(ExpectedConditions.titleIs("Newtab"));

// The rest of your selenium code goes here
```

Handling Popups

- Popup is a window that displays or pops up on the screen due to some activity.
- Popups can be either tooltips on elements or modals that appear on the page as a result of a JS event.
- **Popups are handled the same way as any other HTML element.**
  - **They are NOT the same as alerts.**
- Tooltips usually occur when an element is hovered over.
- Tooltips contain additional helpful information for the user.
- Modals popup on the page as the result of a click event.
- They can contain:
  - Notification Messages
  - Login/Sign up forms
  - Additional information

Example for Handling Popups

```
WebDriver driver = new FirefoxDriver();
// Create the Wait object
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
// Open the page
driver.get("https://training-support.net/webelements/popups");
// Print the title of the page
System.out.println("Page title: " + driver.getTitle());

// Find the launcher button and click it
driver.findElement(By.id("launcher")).click();

// Wait for the modal to appear
wait.until(ExpectedConditions.elementToBeClickable(By.id("username")));

// Find the input fields
WebElement username = driver.findElement(By.id("username"));
WebElement password = driver.findElement(By.id("password"));
// Enter the credentials
username.sendKeys("admin");
password.sendKeys("password");
```

Activity 16

Activity 16

Alerts #1

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/javascript-alerts
- Get the title of the page and print it to the console.
- Find the button to open a **CONFIRM** alert and click it.
- Switch the focus from the main window to the Alert box and get the text in it and print it.
- Close the alert once with **Ok** and again with **Cancel**.
- Close the browser.

Activity16

```
// Set up Firefox driver
        WebDriverManager.firefoxdriver().setup();
        // Create a new instance of the Firefox driver
        WebDriver driver = new FirefoxDriver();

        // Open the page
        driver.get("https://v1.training-support.net/selenium/javascript-alerts");
        // Print the title of the page
        System.out.println("Home page title: " + driver.getTitle());

        // Find and click the button to open the alert
        driver.findElement(By.id("confirm")).click();

        // Switch focus to the alert
        Alert confirmAlert = driver.switchTo().alert();

        // Print the text in the alert
        String alertText = confirmAlert.getText();
        System.out.println("Text in alert: " + alertText);

        // Close the alert by clicking OK
        confirmAlert.accept();

        // Can also close the alert by clicking Cancel
        // confirmAlert.dismiss();

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

Activity 17

Alerts #2

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/javascript-alerts
- Get the title of the page and print it to the console.
- Find the button to open a **PROMPT** alert and click it.
- Switch the focus from the main window to the Alert box and get the text in it and print it.
- Type "Awesome!" into the prompt.
- Close the alert by clicking **Ok**.
- Close the browser.

Activity17

```
// Set up Firefox driver
        WebDriverManager.firefoxdriver().setup();
        // Create a new instance of the Firefox driver
        WebDriver driver = new FirefoxDriver();
        // Create the Wait object
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

        // Open the page
        driver.get("https://v1.training-support.net/selenium/javascript-alerts");
        // Print the title of the page
        System.out.println("Home page title: " + driver.getTitle());

        // Find and click the button to open the alert
        driver.findElement(By.id("prompt")).click();

        // Switch focus to the alert
        Alert promtAlert = driver.switchTo().alert();

        // Print the text in the alert
        String alertText = promtAlert.getText();
        System.out.println("Text in alert: " + alertText);
        // Type into the alert
        promtAlert.sendKeys("Awesome!");

        // Close the alert by clicking OK
        promtAlert.accept();

        // Can also close the alert by clicking Cancel
        // promtAlert.dismiss();

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

Activity 18

Multiple Tabs

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/tab-opener
- Get the title of the page and print it to the console.
- Find the button to open a new tab and click it.
- Wait for the new tab to open.
- Print all the handles.
- Switch to the newly opened tab, print it's title and heading.
- Repeat the steps by clicking the button in the new tab page.
- Close the browser.

Activity 18 Part -1

Activity18

```
// Set up Firefox driver
      WebDriverManager.firefoxdriver().setup();
      // Create a new instance of the Firefox driver
      WebDriver driver = new FirefoxDriver();
      // Create the Wait object
      WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

      // Open the page
      driver.get("https://v1.training-support.net/selenium/tab-opener");
      // Print the title of the page
      System.out.println("Home page title: " + driver.getTitle());

      // Print the handle of the parent window
      System.out.println("Current tab: " + driver.getWindowHandle());
      // Find button to open new tab
      driver.findElement(By.id("launcher")).click();

      // Wait for the second tab to open
      wait.until(ExpectedConditions.numberOfWindowsToBe(2));
      // Print all window handles
      System.out.println("Currently open windows: " + driver.getWindowHandles());

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

Activity 18 Part - 2

Activity18 - 2

```
// Wait for the new page to load
        wait.until(ExpectedConditions.elementToBeClickable(By.id("actionButton")));
        // Print the handle of the current tab
        System.out.println("Current tab: " + driver.getWindowHandle());
        // Print the title and heading of the new page
        System.out.println("Page title: " + driver.getTitle());
        String pageHeading = driver.findElement(By.className("content")).getText();
        System.out.println("Page Heading: " + pageHeading);
        // Find and click the button on page to open another tab
        driver.findElement(By.id("actionButton")).click();

        // Wait for new tab to open
        wait.until(ExpectedConditions.numberOfWindowsToBe(3));
        // Switch focus
        for(String handle : driver.getWindowHandles()) {
            driver.switchTo().window(handle);
        }

        // Wait for the new page to load
        wait.until(ExpectedConditions.visibilityOfElementLocated(By.className("content")));
        // Print the handle of the current tab
        System.out.println("Current tab: " + driver.getWindowHandle());
        // Print the title and heading of the new page
        System.out.println("Page title: " + driver.getTitle());
        pageHeading = driver.findElement(By.className("content")).getText();
        System.out.println("Page Heading: " + pageHeading);

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

Activity 19

Activity 19

Popups

Using Selenium:

- Open a new browser to https://v1.training-support.net/selenium/popups
- Print the title of the page.
- Find the Sign in button and hover over it. Print the tooltip message.
- Click the button to open the Sign in form.
- Enter the credentials
    - username: *admin*
    - password: *password*
- Click login and print the message on the page after logging in.
- Finally, close the browser.

Activity 19 Part -1

Activity19

```
// Set up Firefox driver
WebDriverManager.firefoxdriver().setup();
// Create a new instance of the Firefox driver
WebDriver driver = new FirefoxDriver();
// Create the Wait object
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
// Create the Actions object
Actions builder = new Actions(driver);

// Open the page
driver.get("https://v1.training-support.net/selenium/popups");
// Print the title of the page
System.out.println("Home page title: " + driver.getTitle());

// Find the sign-in button
WebElement button = driver.findElement(By.className("orange"));
// Hover over the button
builder.moveToElement(button).build().perform();
// Print the tooltip message
String tooltipMessage = button.getAttribute("data-tooltip");
System.out.println(tooltipMessage);
```

Activity 19
Part - 2

Activity19 - 2

```
// Click the button and wait for the modal to appear
        button.click();
        wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("username")));

        // Find the input fields
        WebElement username = driver.findElement(By.id("username"));
        WebElement password = driver.findElement(By.id("password"));
        // Enter the credentials
        username.sendKeys("admin");
        password.sendKeys("password");
        // Click the login button
        driver.findElement(By.xpath("//button[text()='Log in']")).click();

        // Print the login message
        String message = driver.findElement(By.id("action-confirmation")).getText();
        System.out.println("Login message: " + message);

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

Browser Configuration

Browser Configuration

- Browser configurations and desired capabilities in Selenium are mechanisms to customize browser behavior and specify requirements for test execution.
- Browser options classes in Selenium define capabilities shared by all browsers.
- WebDriver capabilities are used to communicate(set) the features supported by a given implementation. The local end(client side) may use capabilities to define which features it requires the remote end(server side) to satisfy when creating a new session.
- In addition to standard WebDriver capabilities, Selenium provides a way to Proxy settings.

References:
Browser Options
Capabilities
Browser Specific capabilities

Browser Configuration

Why Configure Browsers?

- **Customization:** Tailor browser behavior for specific testing needs.
- **Environment Control:** Ensure tests run consistently across different setups.
- **Performance/Resource Management:** Run tests efficiently (e.g., headless mode).
- **Debugging:** Set specific user agents, proxies, or logging levels.
- **Real-world Scenarios:** Simulate specific user environments (e.g., mobile view).

Browser Configuration

DesiredCapabilities

- A set of **key-value pairs** used to define **what kind of browser and environment** you want to test against.
- Think of it as a "wish list" sent to the WebDriver server.
- To request specific browser features, versions, platforms, or behaviors (e.g., accepting insecure SSL certs, proxy settings).
- Historically, this was the main way to configure browser settings and capabilities in Selenium.
- Today, for modern **local** browser automation, it’s mostly replaced by browser-specific **Options** classes (like `ChromeOptions` and `FirefoxOptions`).

Browser Configuration

However, DesiredCapabilities is still important when:

- Working with **Selenium Grid**, to communicate test requirements (browser, version, platform) to remote nodes.
- Running tests on **cloud testing services** (like BrowserStack or Sauce Labs).
- Dealing with less common or older browsers that don’t have dedicated Options classes.

...

```
//package
import org.openqa.selenium.remote.DesiredCapabilities;
  // Create DesiredCapabilities object
        DesiredCapabilities caps = new DesiredCapabilities();

        // Set browser name and platform
        caps.setBrowserName("chrome");
        caps.setPlatform(org.openqa.selenium.Platform.WINDOWS);

        // Accept insecure SSL certificates (e.g., for testing dev environments)
        caps.setAcceptInsecureCerts(true);

        // Example: Setting a custom capability (might be used by Grid or a cloud provider)
        caps.setCapability("customCapabilityName", "customValue");

        // How to use it (Legacy/Remote WebDriver):
        // WebDriver driver = new ChromeDriver(caps); // Note: This constructor might be
deprecated or behave differently for local
        // WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444/wd/hub"),
caps); // For Selenium Grid
```

Browser Configuration

Modern Browser Configuration: Options Classes

- The Preferred Way for Local Browsers
  - Browser-specific `Options` classes (e.g., `ChromeOptions`, `FirefoxOptions`, `EdgeOptions`).
  - Offer a more **type-safe, intuitive, and feature-rich** way to configure modern browsers locally.
  - Each `Options` class provides methods specific to its browser (e.g., Chrome-specific arguments).

Browser Configuration

Common Configurations using `Options`:

- **Headless Mode:** Run browser without a visible UI (faster, good for CI/CD).
- **Start Maximize:** Start the browser in maximized window state.
- **User Agent:** Change the browser's reported user agent string.
- **Extensions/Add-ons:** Load custom browser extensions.
- **Arguments:** Pass command-line arguments to the browser executable.
- **Experimental Options:** Access browser-specific experimental features.

...

```
ChromeOptions options = new ChromeOptions();

        //Add some arguments (these are optional)
        options.addArguments("--start-maximized");    // Start Chrome maximized
        options.addArguments("--incognito");          // Open in Incognito mode
        options.addArguments("--disable-extensions"); // Disable extensions

        //Initialize ChromeDriver with options
        WebDriver driver = new ChromeDriver(options);

        //Do something with the browser
        driver.get("https://www.google.com");

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

Chrome Options

Chrome Options

```
ChromeOptions chromeOptions = new ChromeOptions();

// Common options
chromeOptions.addArguments("--start-maximized");           // Maximize browser
chromeOptions.addArguments("--disable-notifications");     // Disable notifications
chromeOptions.addArguments("--incognito");                 // Incognito mode
chromeOptions.addArguments("--headless=new");              // Headless mode
chromeOptions.addArguments("--disable-extensions");        //Disable all extensions
chromeOptions.setAcceptInsecureCerts(true);   //Accept self-signed SSL certificates
chromeOptions.setPageLoadStrategy(PageLoadStrategy.EAGER); // Page load strategy

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

FireFox Options

Fire fox

```
FirefoxOptions firefoxOptions = new FirefoxOptions();

// Common options
firefoxOptions.addArguments("-private");                // Private mode
firefoxOptions.addArguments("-headless");               // Headless mode
firefoxOptions.setAcceptInsecureCerts(true); // Accept self-signed SSL certificates
firefoxOptions.setPageLoadStrategy(PageLoadStrategy.EAGER); // Page load strategy

// Disable notifications
firefoxOptions.addPreference("dom.webnotifications.enabled", false);

// Disable extensions (for Firefox, use 'xpinstall.signatures.required' and
'extensions.enabled')
firefoxOptions.addPreference("xpinstall.signatures.required", false);
firefoxOptions.addPreference("extensions.enabled", false);

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

Edge Options

Edge options

```
EdgeOptions edgeOptions = new EdgeOptions();

// Common options
edgeOptions.addArguments("--start-maximized");           // Maximize browser
edgeOptions.addArguments("--disable-notifications");     // Disable notifications
edgeOptions.addArguments("--inprivate");                 // Private mode
edgeOptions.addArguments("--headless=new");              // Headless mode
edgeOptions.addArguments("--disable-extensions");        // Disable extensions
edgeOptions.setAcceptInsecureCerts(true);    // Accept self-signed SSL certificates
edgeOptions.setPageLoadStrategy(PageLoadStrategy.EAGER); // Page load strategy

WebDriver driver = new EdgeDriver(edgeOptions);
```

Table of standard capabilities

Capability	Key	Value Type	Description
Browser name	`"browserName"`	string	Identifies the user agent.
Browser version	`"browserVersion"`	string	Identifies the version of the user agent.
Platform name	`"platformName"`	string	Identifies the operating system of the endpoint node.
Accept insecure TLS certificates	`"acceptInsecureCerts"`	boolean	Indicates whether untrusted and self-signed TLS certificates are implicitly trusted on navigation for the duration of the session.
Page load strategy	`"pageLoadStrategy"`	string	Defines the session's page load strategy.
Proxy configuration	`"proxy"`	JSON Object	Defines the session's proxy configuration.
Window dimensioning/ positioning	`"setWindowRect"`	boolean	Indicates whether the remote end supports all of the resizing and repositioning commands.
Session timeouts	`"timeouts"`	JSON Object	Describes the timeouts imposed on certain session operations.
Strict file interactability	`"strictFileInteractability"`	boolean	Defines the session's strict file interactability.
Unhandled prompt behavior	`"unhandledPromptBehavior"`	string	Describes the session's user prompt handler. Defaults to `"dismiss and notify"`.
User Agent	`"userAgent"`	string	Identifies the default User-Agent value of the endpoint node.

Activity 20

Activity 20

Using Selenium:

- Open a new Chrome browser to `https://expired.badssl.com/`
- Print the title of the page.
- Accept the insecure SSL certificate (handled by `ChromeOptions`).
- Start the browser in Incognito mode.
- Open the browser maximized.
- Assert the title of the page
- Finally, close the browser.

Activity20

```
ChromeOptions options = new ChromeOptions();
        //Accept insecure certs
        options.setAcceptInsecureCerts(true);
        //Start maximized
        options.addArguments("--start-maximized");
        //Open incognito
        options.addArguments("--incognito");
        //Create web driver object
        WebDriver driver = new ChromeDriver(options);

        driver.get("https://expired.badssl.com/");

        System.out.println(driver.getTitle());

        Assert.assertEquals(driver.getTitle(),"expired.badssl.com" );

        driver.quit();
```

JavascriptExecutor

JavascriptExecutor

- The `JavaScriptExecutor` provides an interface that enables testers to run JavaScript methods from Selenium scripts.
- It acts as a medium that enables the WebDriver to interact directly with HTML elements within the browser using JavaScript.
- It helps Selenium test scripts to run as desired by overcoming issues that arise due to **browser incompatibility** or when standard WebDriver commands face problems.
- `JavascriptExecutor` comes into the picture when Selenium WebDriver encounters problems interacting with certain web elements.
- **For instance:** If an unexpected pop-up prevents WebDriver from locating a specific element, `JavascriptExecutor` can bypass such issues to produce accurate results.

JavascriptExecutor

`JavascriptExecutor` consists of methods that handle all essential interactions:

- **`executeScript()` (Synchronous):**
  a. **Running Immediate JavaScript Commands**
     i. Executes JavaScript code directly in the browser and **waits for the script to finish** before proceeding with your Selenium test.
     ii. Giving an immediate command to the browser.
     iii. For most common `JavascriptExecutor` tasks where you need the result or action to complete right away.

Syntax

executeScript syntax

```
// Cast WebDriver to JavascriptExecutor
JavascriptExecutor js = (JavascriptExecutor) driver;

// Method signature
js.executeScript(String script, Object... args);
```

Usage of JavascriptExecutor

Example

```
JavascriptExecutor js = (JavascriptExecutor) driver;
js.executeScript("arguments[0].click();", element);
js.executeScript("window.scrollBy(0,500);");
js.executeScript("arguments[0].value='text';", inputElement);
```

Activity 21

Activity 21

Using Selenium:

- Open a new Firefox browser and navigate to **https://en.wikipedia.org/wiki/Artificial_intelligence**
- Scroll to the footer using **JavaScriptExecutor**.
- Find the **“About Wikipedia”** link inside the footer.
- Click the **“About Wikipedia”** link using **JavaScriptExecutor**.
- Assert that page title
- Finally, close the browser.

Activity21

```
 WebDriver driver = new FirefoxDriver();
        driver.get("https://en.wikipedia.org/wiki/Artificial_intelligence");

        WebDriverWait wait = new WebDriverWait(driver, Duration.ofMillis(5000));
        WebElement footer =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("footer")));
        WebElement aboutWikipedia =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.cssSelector("#footer-places-about
> a")));

        JavascriptExecutor js = (JavascriptExecutor) driver;
        //Scroll to the footer
        js.executeScript("arguments[0].scrollIntoView(true);", footer);
        //Click on the aboutWikipedia
        js.executeScript("arguments[0].click();", aboutWikipedia);

        String title = driver.getTitle();
        System.out.println("Page title: " + title);

        Assert.assertEquals(title, "Wikipedia:About - Wikipedia");

        driver.close();
```

Common Exceptions

Exception Handling in Selenium

- An exception is an incident that disturbs the normal flow of a program's execution. When an unexpected occurrence occurs during the execution of a program, an exception object is created to reflect the specific error or unexpected state.
- An exception can be caused due to: logical errors, runtime errors and system errors.
- To properly handle exceptions, programming provide methods such as try-except blocks (try-catch in some), in which code that may cause an exception is trapped within a try block and potential exceptions are caught and handled in except blocks. This helps to gracefully manage failures and keep the software stable.

Benefits of Exception Handling

Here are the key reasons why exception handling is important:

- Prevents abrupt test failures by catching errors.
- Gracefully handles unexpected issues like timeouts and missing elements.
- Simplifies debugging with clear error feedback.
- Enhances overall test stability and reliability.
- Supports consistent performance in CI/CD environments.

Common Exceptions in Selenium WebDriver

Exception	Cause	Solution
NoSuchElement Exception	Looking for element in wrong place/ at wrong time. The locator has changed since the time we wrote script	Make sure that you are on correct website and previous actions in script are completed. Use proper waiting strategies and browser devtools to update locator.
NoAlertPresent Exception	Raised when an expected alert is not present. Happens when trying to interact with an alert that doesn't exist.	Check if the alert is present before trying to interact with it

Common Exceptions in Selenium WebDriver

Exception	Cause	Solution
StaleElementReferenceException	DOM was refreshed (page reload or navigate back/forward) - Element was removed and reinserted (`outerHTML` or dynamic rendering) - AJAX content updated part of the page	lways **re-find the element** before interacting if you suspect the DOM has changed. - Use a `try-catch` block to catch the exception and retry by locating the element again. - Use proper waits to ensure content has loaded.
NoSuchWindow Exception	While attempting to access a non-existent browser window during test execution.	Avoid Accessing Closed Windows Provide a Valid Window Handle Use Wait Methods

Other Common Exceptions in Selenium WebDriver

- **MoveTargetOutOfBoundsException**: occurs when attempting to interact with an element that is not within the viewable area of the browser.
- **InvalidArgumentException**: Thrown when an argument passed to a method is invalid. Occurs when incorrect arguments are provided, such as invalid locators or options.
- **UnhandledAlertException**: Raised when an unexpected alert is present. Commonly occurs when an alert appears, but the script doesn't handle it, and further actions fail.
- **JavascriptException**: Thrown when executing JavaScript through Selenium fails.

Some New Exceptions in Selenium 4.0

- **ElementClickInterceptedException**: is raised when an element you try to click is not clickable because another element is blocking it.
- **NoSuchCookieException**: Raised when trying to interact with a cookie that doesn’t exist.
- **InvalidCoordinatesException**: Raised when there is an issue with the coordinates provided for actions like mouse movements.
- **InvalidSessionIdException**: Raised when the session ID used in a request is invalid or has expired.
- **ElementNotInteractableException**:is raised when an element is found but is not interactable (e.g., it’s hidden or disabled).

Activity 22

Activity 22

In this activity, you will learn to handle `StaleElementReferenceException` in Selenium.

Using Selenium:

- Open Firefox and go to: https://v1.training-support.net/selenium/ajax
- Locate and **click** the **"Change Content"** button
- Wait for the **new heading text** (<h3>) that appears dynamically after AJAX loads.
- Print the text to the console.
- **Navigate away** to https://v1.training-support.net/selenium.
- Wait 2 seconds and **navigate back**.
- Try to click the **"Change Content"** button again
- Close the browser

Activity22

```
    WebDriver driver = new FirefoxDriver();
        driver.get("https://v1.training-support.net/selenium/ajax");

        WebElement btn = driver.findElement(By.cssSelector(".ui.violet"));
        btn.click();

        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));
        WebElement headingTag =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//h3[contains(text(),
'late')]")));
        System.out.println(headingTag.getText());
        //Navigate to different page
        driver.navigate().to("https://v1.training-support.net/selenium");
        Thread.sleep(2000);
        //navigate back to the Ajax page
        driver.navigate().back();

        //This will cause StaleElementReferenceException!
        btn.click();
```

Did you get `StaleElementReferenceException`? Good! Now handle it

Use Try-catch blocks

Activity22

```
WebDriver driver = new FirefoxDriver();
        driver.get("https://v1.training-support.net/selenium/ajax");

        WebElement btn = driver.findElement(By.cssSelector(".ui.violet"));
        btn.click();

        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));
        WebElement headingTag =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//h3[contains(text(),
'late')]")));
        System.out.println(headingTag.getText());

        driver.navigate().to("https://v1.training-support.net/selenium");
        Thread.sleep(2000);
        driver.navigate().back();

        try {
            // This may be stale
            btn.click();
        } catch (StaleElementReferenceException s) {
            // Re-locate and retry
            btn = driver.findElement(By.cssSelector(".ui.violet"));
            btn.click();
        } finally {
            //close the browser
            driver.close();
        }
```

Activity 23

Activity 23

In this activity, you will learn to work with Forms, Sliders, and Screenshots

- Open the browser and navigate to **https://training-support.net/complex-forms/catering**
- Fill **all the fields** in the form.
- Set the **Number of Guests** slider to `100`.
- Take a **screenshot** of the filled form.
- Submit the form.
- Verify that the confirmation message says **“Booking Successful!”**
- Close the browser.

Activity23 - 1

```
WebDriver driver = new FirefoxDriver();
    driver.get("https://training-support.net/complex-forms/catering");
    driver.manage().window().maximize();

    // Fill in form fields
    driver.findElement(By.id("name")).sendKeys("John");
    driver.findElement(By.id("phone")).sendKeys("9876543210");
    driver.findElement(By.id("address")).sendKeys("Hyderabad");

    // Select catering type
    WebElement selectElement = driver.findElement(By.id("catering-type"));
    Select sel = new Select(selectElement);
    sel.selectByIndex(3);
```

Activity23 - 2

```
// Adjust slider using JS
    WebElement slider = driver.findElement(By.id("num-guests"));
    JavascriptExecutor js = (JavascriptExecutor) driver;
    js.executeScript("arguments[0].value = 100", slider);
    js.executeScript("arguments[0].dispatchEvent(new Event('input', {bubbles: true}))",
slider);

    // Fill date and time
    driver.findElement(By.id("event-date")).sendKeys("2025-07-10");
    driver.findElement(By.id("event-time")).sendKeys("19:00");

    // Take screenshot
    TakesScreenshot shot = (TakesScreenshot) driver;
    File screenshot = shot.getScreenshotAs(OutputType.FILE);
    File dest = new File("C:/Users/Dilip/Desktop/slider.png");
    FileUtils.copyFile(screenshot, dest);

    // Submit form and check result
    driver.findElement(By.xpath("//button[text() = 'Submit']")).click();
    String message = driver.findElement(By.id("action-confirmation")).getText();
    Assert.assertEquals(message, "Booking Successful!");

    driver.quit();
```

Activity 24

Activity 24

- Open the browser and navigate to https://training-support.net/complex-forms/job-application
- Fill **all required fields** in the form:
  - Name
  - Email
  - Phone number
  - Position
  - Employment status
  - Upload a resume file
  - Submit the form.
- Verify that the confirmation message says **`Application Submitted Successfully!`**
- Close the browser.

Activity24

```java
// Launch Chrome browser
WebDriver driver = new ChromeDriver();
// Open the Job Application form page
driver.get("https://training-support.net/complex-forms/job-application");

// Fill in basic details
driver.findElement(By.id("name")).sendKeys("Sam Hain");
driver.findElement(By.id("email")).sendKeys("samhain@gmail.com");
driver.findElement(By.id("phone")).sendKeys("123456789");

// Select position from dropdown
WebElement selectElement = driver.findElement(By.id("position"));
Select select = new Select(selectElement);
select.selectByValue("Full Stack Developer");

driver.findElement(By.xpath("//input[@value = 'Employed']")).click();

// Upload resume fille
driver.findElement(By.id("resume")).sendKeys("//absolute path to file");

driver.findElement(By.xpath("//button[text() = 'Submit']")).click();

// Verify confirmation message
String message = driver.findElement(By.id("action-confirmation")).getText();
Assert.assertEquals(message, "Application Submitted Successfully!");

// Close the browser
driver.close();
```

Thank You