
Se
Selenium

Course Content

- What is Selenium
- Selenium Tool Suite
- Selenium IDE, RC, WebDriver, Grid
- Working with WebElements
- Locator Strategies
- Actions on WebElements
- Different Interactions on Webpages
- Handling Exceptions
- Waiting Mechanisms
- JavaScript Executor
- Many More …




What is Selenium ?

- Selenium is a powerful, open-source suite for automating web applications.
- Supports a wide range of browsers (**Chrome, Firefox, Safari, etc.**) and platforms (**Windows, macOS, Linux**).
- Enables testing in multiple programming languages (**Java, Python, C#, etc.**).
- Highly extensible and widely used in DevOps and CI/CD pipelines.

Image Source: *https://x.com/testingb0t/status/1851292815390916736*

Why Selenium ?

- Open-source and free.
- Cross-browser and cross-platform compatibility.
- Extensive community support.
- Flexible integration with tools like TestNG, JUnit, and CI/CD pipelines (e.g., Jenkins).
- Versatile for functional, regression, and end-to-end testing.

Components of Selenium Suite

1. Selenium IDE:

- A record-and-playback tool for creating simple test scripts.
- Best for beginners and quick prototyping.

2. Selenium RC (Remote Control):

- Legacy tool for automating web testing using servers.
- Deprecated in favor of WebDriver.

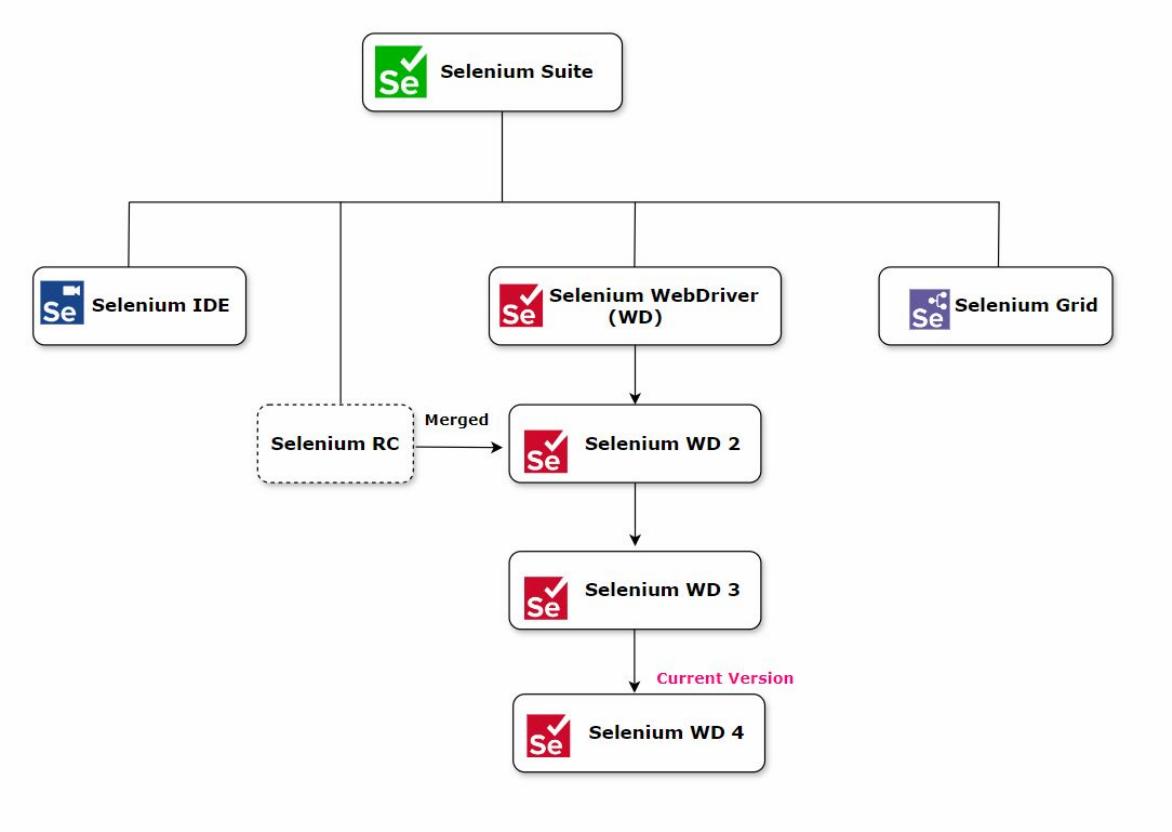
3. Selenium WebDriver:

- Core framework for advanced web testing.
- Directly interacts with the browser without a server.

4. Selenium Grid:

- Enables parallel test execution across multiple environments.
- Speeds up testing for large-scale systems.

Selenium Suite Overview



Selenium IDE

Key Features:

- Browser extension (available for Chrome and Firefox).
- Record-and-playback functionality for rapid test creation.
- Generates scripts in various languages.

Limitations:

- Not suitable for complex test scenarios.
- Limited support for conditional logic and loops.

Ideal for: Quick and simple automation tasks.

Selenium RC

Allowed testing in multiple languages by interacting with the browser through a server.

Disadvantages:

- Slower execution due to server dependency.
- Outdated with the release of WebDriver.

Status: Deprecated (Replaced by WebDriver).

Selenium Webdriver

Key Features:

- Programmatic control of browsers without requiring a server.
- Supports advanced automation scenarios (e.g., dynamic web elements, AJAX).
- Supports most modern browsers and mobile platforms.

Why WebDriver?

- Faster, more reliable, and flexible compared to Selenium RC.
- Direct API access for precise browser control.

Selenium Grid

Purpose: Parallel test execution across multiple devices, browsers, and OS.

How It Works:

- A central hub manages the distribution of test cases to nodes (machines/browsers).

Key Benefit: Saves time by running tests simultaneously.

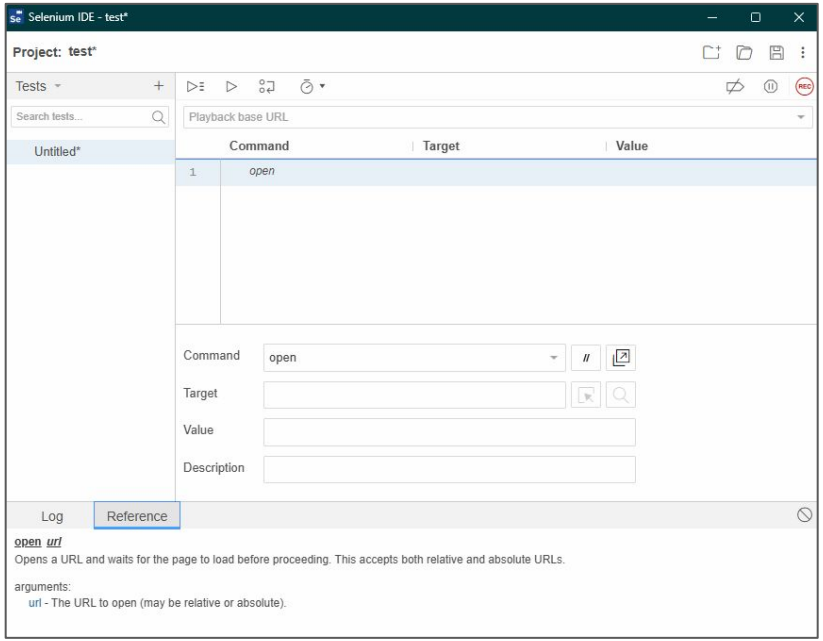
Ideal for: Large test suites and cross-browser compatibility testing.

Comparison of Selenium Tools

Feature	Selenium IDE	Selenium RC (Legacy)	Selenium WebDriver	Selenium Grid
Ease of Use	High	Moderate	Moderate	Moderate
Flexibility	Low	High	High	High
Parallel Testing	No	No	No (partial)	Yes
Current Relevance	Yes	No (Deprecated)	Yes	Yes

Selenium IDE

- Selenium IDE is a browser extension for Chrome and Firefox.
- Used for recording, editing, and playing back functional test cases.
- Simplifies test creation for beginners or quick prototyping.
- Generates scripts in multiple languages like Java, Python, and JavaScript.



Selenium IDE

Advantages:

- User-friendly interface with zero coding required for basic tests.
- Accelerates test case creation with record-and-playback.
- Supports exporting test cases to programming languages for advanced use.
- Allows integration with CI/CD pipelines.

Limitations:

- Not suitable for complex test scenarios.
- Limited support for conditional logic, loops, and advanced error handling.

Use Case: Rapid prototyping and quick automation validation.

Installing Selenium IDE

Step 1: Go to the official Selenium IDE download page.

https://www.selenium.dev/selenium-ide/

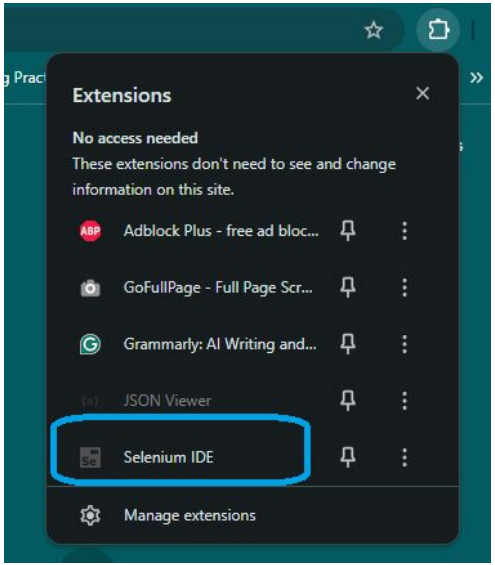
Step 2: Install the browser extension for Chrome or Firefox.

https://chromewebstore.google.com/detail/selenium-ide/mooikfkahbdckldjjndioackbalphokd

https://addons.mozilla.org/en-GB/firefox/addon/selenium-ide/

Step 3: Confirm installation and enable the extension.

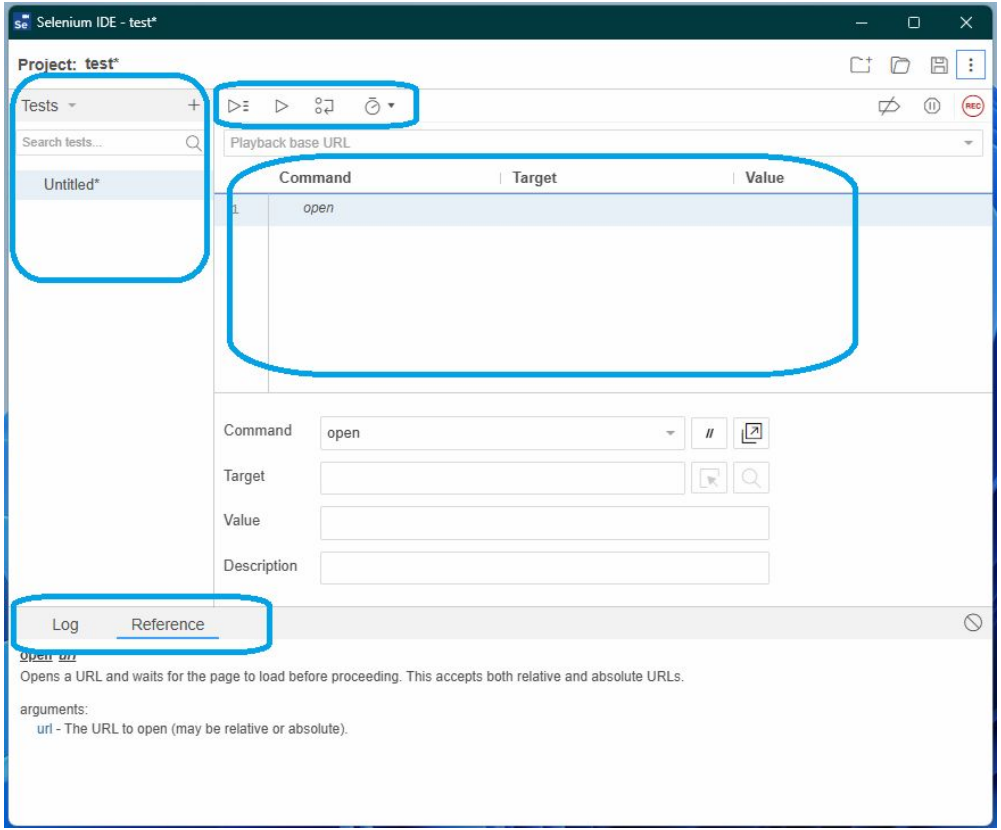
Step 4: Open Selenium IDE from the browser toolbar.



Selenium IDE Interface Overview

Main Components:

1. **Menu Bar:** Options for creating, saving, and exporting test cases.
2. **Test Case Pane:** Displays recorded steps and actions.
3. **Toolbar:** Buttons for record, playback, and step execution.
4. **Log/Reference Pane:** Shows execution logs and command details.



Creating First Test Case

Steps to Create a Test Case:

1. Click on the **Record** button to start recording.
2. Perform actions in the browser (e.g., navigate, click, type).
3. Stop recording when done.
4. Save the test case with a meaningful name.

Selenium IDE will capture commands like:

- open: Navigates to a URL.
- click: Simulates a mouse click.
- type: Enters text into input fields.

Running and Debugging a Test

Running Tests:

- Use the Play button to execute all steps.
- Use the Step feature to execute commands one-by-one.

Debugging:

- Review the Log pane for errors.
- Use breakpoints to pause execution for troubleshooting.
- Edit the test case to fix issues and rerun.

Exporting Test Cases

Export recorded tests to programming languages for advanced usage.

Supported languages include:

- Java (TestNG or JUnit)
- Python (Pytest)
- JavaScript (Node.js)

Steps to Export:

1. Open the test case in Selenium IDE.
2. Click on **Export** from the menu.
3. Select the desired format and save the file.

Ideal for integrating into a larger automation framework.

Common Selenium IDE Commands

Command	Description	Example
open	Opens a URL in the browser	`open https://google.com`
click	Clicks on a specified element	`click id=submit`
type	Inputs text into a field	`type id=username`
verifyText	Checks if a text is present on the page	`verifyText Welcome`
waitFor	Waits for an element to appear	`waitFor id=loader`

Advanced Features

1. **Conditional Statements:**
    - Use if, elseIf, and else commands for decision-making.
2. **Loops:**
    - Repeat actions using doRepeatIf and times.
3. **Assertions:**
    - Verify application behavior with assert commands.
4. **Plugins:**
    - Extend functionality with third-party plugins.

Introduction to Selenium WebDriver

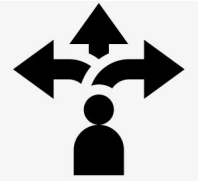
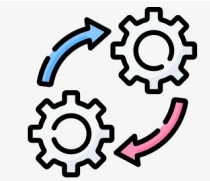
- A core component of the Selenium suite designed for browser automation.
- Allows direct communication with browsers via native APIs.
- Supports dynamic and interactive web elements, making it suitable for testing modern web applications.
- https://www.selenium.dev/documentation/webdriver/

Key Features of Selenium WebDriver

- **Browser Control:** Works directly with browser drivers for better performance and reliability.
- **Multi-Language Support:** Write scripts in Java, Python, C#, JavaScript, etc.
- **Cross-Browser Compatibility:** Automates Chrome, Firefox, Edge, Safari, and more.
- **Support for Dynamic Elements:** Handles AJAX, iframes, and other dynamic components effectively.
- **No Server Dependency:** Eliminates the server layer (unlike Selenium RC).

Why Choose Selenium WebDriver? - Advantages

- **Fast Execution:** No intermediate server like Selenium RC.
- **Scalable:** Works seamlessly with Selenium Grid for parallel execution.
- **Customizable:** Supports advanced frameworks and libraries.
- **Extensible:** Integrates with tools like TestNG, JUnit, and CI/CD pipelines.
- **Mobile Testing Support:** Automates browsers on mobile devices.


Limitations of Selenium WebDriver?

- Requires programming knowledge for script creation.
- Limited to web applications (does not support desktop or API testing).
- Debugging can be complex for large test suites.
- Manual setup required for browser drivers.

How WebDriver Works ?

- Uses browser-specific drivers to send commands to the browser.
- Drivers act as a bridge between the script and the browser.
- Example drivers:
  - chromedriver for Chrome
  - geckodriver for Firefox
  - msedgedriver for Edge
- Executes commands like navigation, input simulation, and assertions.

Selenium WebDriver Installation & Setup

Step 1: Install Prerequisites

- Java Development Kit (JDK).
- An IDE (e.g., IntelliJ IDEA, Eclipse).
- Maven for dependency management.

Step 2: Create a Maven Project

- Open your IDE and create a new Maven project.
- Define the groupId, artifactId, and project name.

Step 3: Add Selenium Dependencies

- Open the pom.xml file in your project.

```xml
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.x.x</version>
</dependency>
```

First WebDriver Script

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

public class FirstTest {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        driver.get("https://www.saucedemo.com/");
        System.out.println("Page title is: " + driver.getTitle());
        driver.quit();
    }
}
```

Web Fundamentals

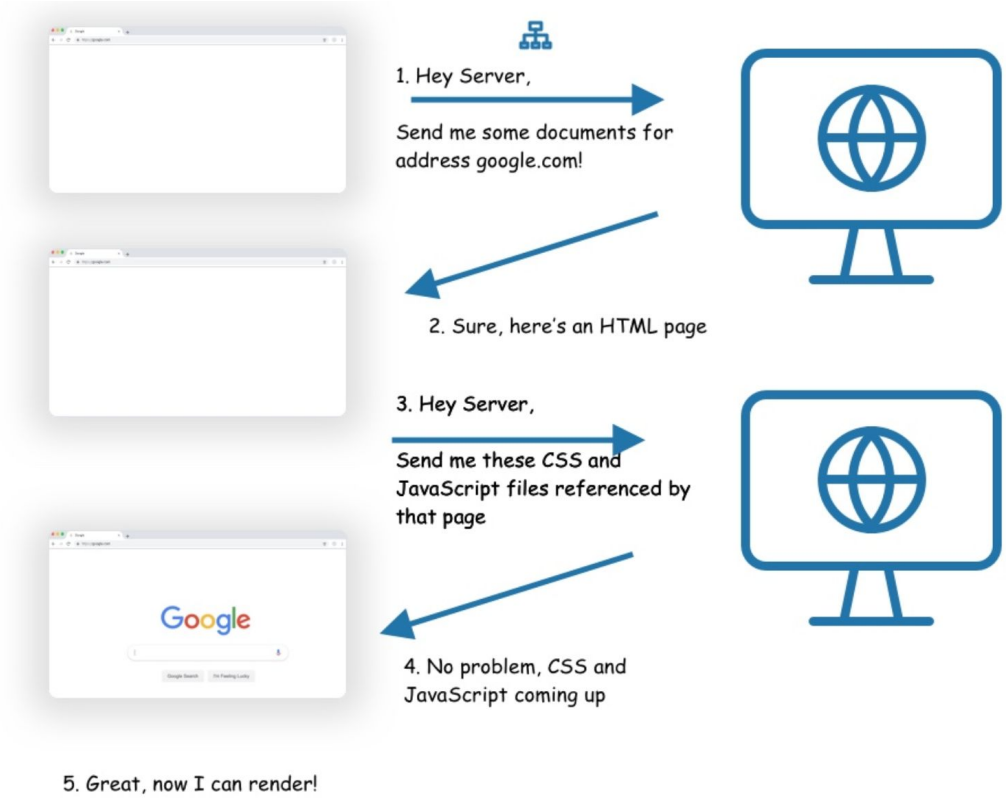


Image Source: *https://zendev.com/2019/04/02/how-do-web-applications-actually-work.html*

Frontend

Focuses on layout, animations, content organization, navigation, graphics

Programming Languages:

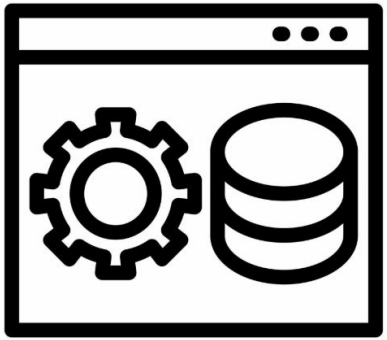
JavaScript, HTML, CSS

Backend

Focuses on building code, debugging, database management.

Programming Languages:

NodeJS, Python, Java



What’s the difference ?

HTML

Hypertext Markup Language

Create the structure

- Controls the layout of the content
- Provides structure for the web page design
- The fundamental building block of any web page

CSS

Cascading Style Sheet

Stylize the website

- Applies style to the web page elements
- Targets various screen sizes to make web pages responsive.
- Primarily handles the look and feel of a web page

JS

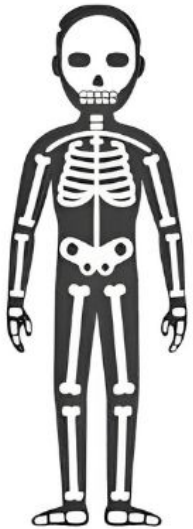
Javascript

Increase interactivity

- Adds interactivity to a web page
- Handles complex functions and features
- Programmatic code which enhances functionality

HTML

HTML the Skeleton

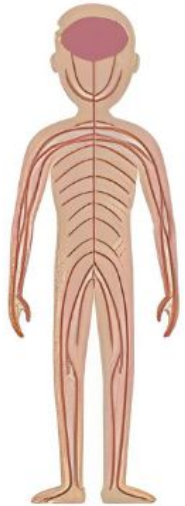


CSS

CSS the Skin

JS

Javascript the Brain



HTML Basics

- HTML stands for **HyperText Markup Language**.
- It’s the standard language for structuring content on the web.
- HTML documents consist of **tags** and **attributes** that define the layout and functionality of webpages.

Basic Structure of an HTML Document

1. **<html>**: The root element that encloses all HTML content.
2. **<head>**: Contains metadata like title and styles.
3. **<body>**: Contains the visible content of the webpage.

Common HTML Tags

Structural Tags: Define page layout

- : Generic container
- <section>: Groups related content
- <header>, <footer>: Top and bottom sections

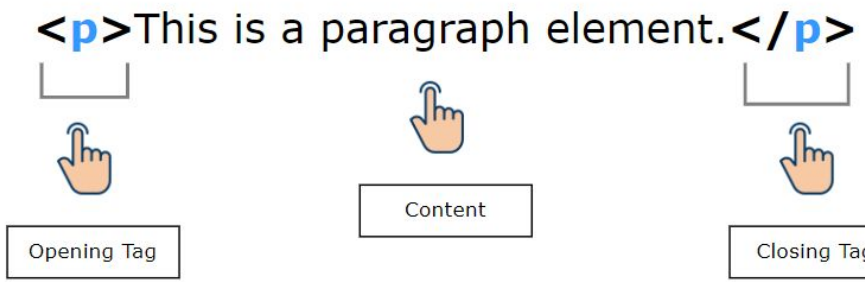
Text Content Tags: Display text

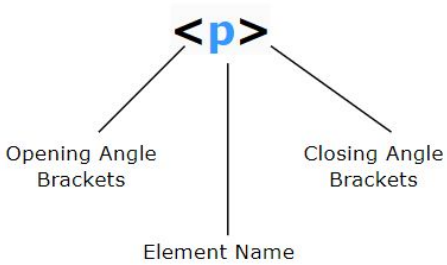
- <h1> to <h6>: Headings
- <p>: Paragraph
- : Inline container

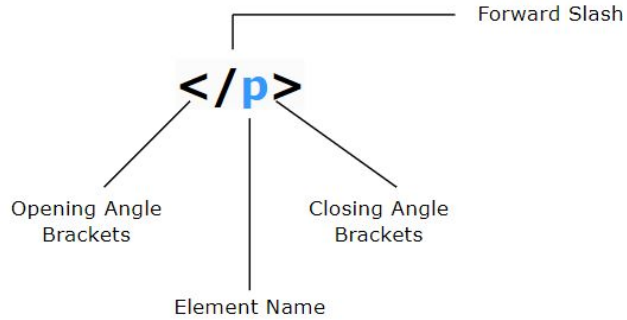
Interactive Tags: Add interactivity

- <a>: Hyperlinks
- <button>: Buttons
- <input>: User input fields

HTML Element







HTML Attributes

Attributes provide additional details about HTML elements.

Common attributes:

- id: Unique identifier.
- class: Groups elements for styling or scripting.
- name: Often used in forms for data submission.
- value: Specifies the value of an element.

For Example:

```
<input type="text" id="username" name="user" placeholder="Enter your username">
```

type="text": Input field type.

id="username": Unique identifier.

name="user": Name for form submission.

placeholder="Enter your username": Hint text.

Commonly used HTML tags

Tag	Description	Example
<html>	Root element of an HTML document	<html><head></head><body></body></html>
<head>	Metadata about the HTML document	<head><title>Page Title</title></head>
<title>	Specifies the document title	<title>My Web Page</title>
<body>	Main content of the HTML document	<body> Hello, World! </body>
<h1> to **<h6>**	Headings, <h1> is the largest, <h6> is the smallest	Main Heading Sub Heading
** **	Paragraph text	This is a paragraph.
<a>	Hyperlink to another page or section	Visit Example
<img>	Embeds an image	<img src="image.jpg" alt="Image description">
** **	Unordered (bulleted) list	- Item 1 - Item 2
** **	Ordered (numbered) list	- First - Second

Commonly used HTML tags Contd.

Tag	Description	Example
** **	List item	List Item
** **	A block-level container	Content inside a div
****	An inline container	This is red text.
<table>	Table structure	<table><tr><td>Cell</td></tr></table>
<tr>	Table row	<tr><td>Row Data</td></tr>
<td>	Table data cell	<td>Table Cell</td>
<th>	Table header cell	<th>Header</th>
<form>	Form for user input	<form><input type="text" name="name"><button>Submit</button></form>
<input>	Input field	<input type="text" placeholder="Enter text">
<button>	Button element	<button>Click Me</button>

Commonly used HTML tags Contd.

Tag	Description	Example
<select>	Dropdown list	<select><option>Option 1</option><option>Option 2</option></select>
<option>	Dropdown list option	<option>Option</option>
<label>	Label for input fields	<label for="name">Name:</label><input id="name" type="text">
<textarea>	Multi-line text input	<textarea rows="4" cols="50">Enter text here</textarea>
<iframe>	Embeds another page	<iframe src="https://example.com" width="300" height="200"></iframe>
****	Bold text	Important Text
****	Italicized (emphasized) text	Emphasized Text
** **	Line break	Line 1 Line 2
<hr>	Horizontal line	<hr>
<meta>	Metadata about the page	<meta charset="UTF-8">

Commonly used HTML tags Contd.

Tag	Description	Example
<link>	Link to external resources (e.g., CSS)	<link rel="stylesheet" href="styles.css">
<script>	Embeds JavaScript code	<script>console.log('Hello World');</script>
<style>	Embeds CSS styles	<style>body { background-color: lightgray; }</style>
<header>	Header section of a page	<header> Website Name </header>
<footer>	Footer section of a page	<footer>© 2024 My Website</footer>
<nav>	Navigation links	<nav>HomeAbout</nav>
<section>	Defines a section in a document	<section> About Us Details about us. </section>

The DOM (Document Object Model)

The **Document Object Model (DOM)** is a programming interface for web documents. It represents the structure of a webpage as a tree of objects that can be manipulated with programming languages like **JavaScript**. This allows developers to dynamically update content, structure, and styles of a webpage without reloading it.

Definition:

- The DOM is a **tree-like representation** of the HTML document.
- It is a standard defined by the **W3C (World Wide Web Consortium)**.

Role in Web Development:

- Allows interaction with the webpage programmatically.
- Enables dynamic content updates (e.g., showing/hiding elements, fetching and displaying data).

Tree Structure:

- The HTML document is modeled as a tree of **nodes**:
  - The root node is `<html>`.
  - Other nodes include `<head>`, `<body>`, and their child elements.

DOM Structure Overview

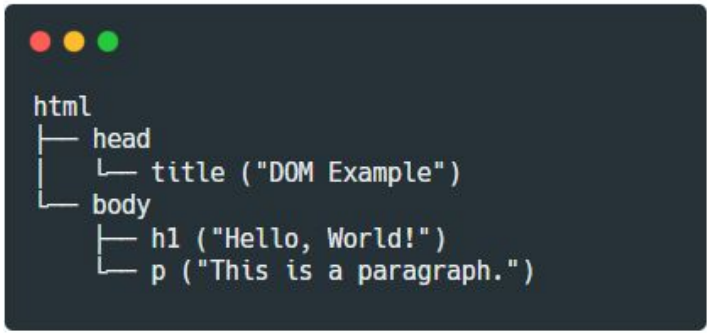
The DOM represents each element, attribute, and text in a document as a node.

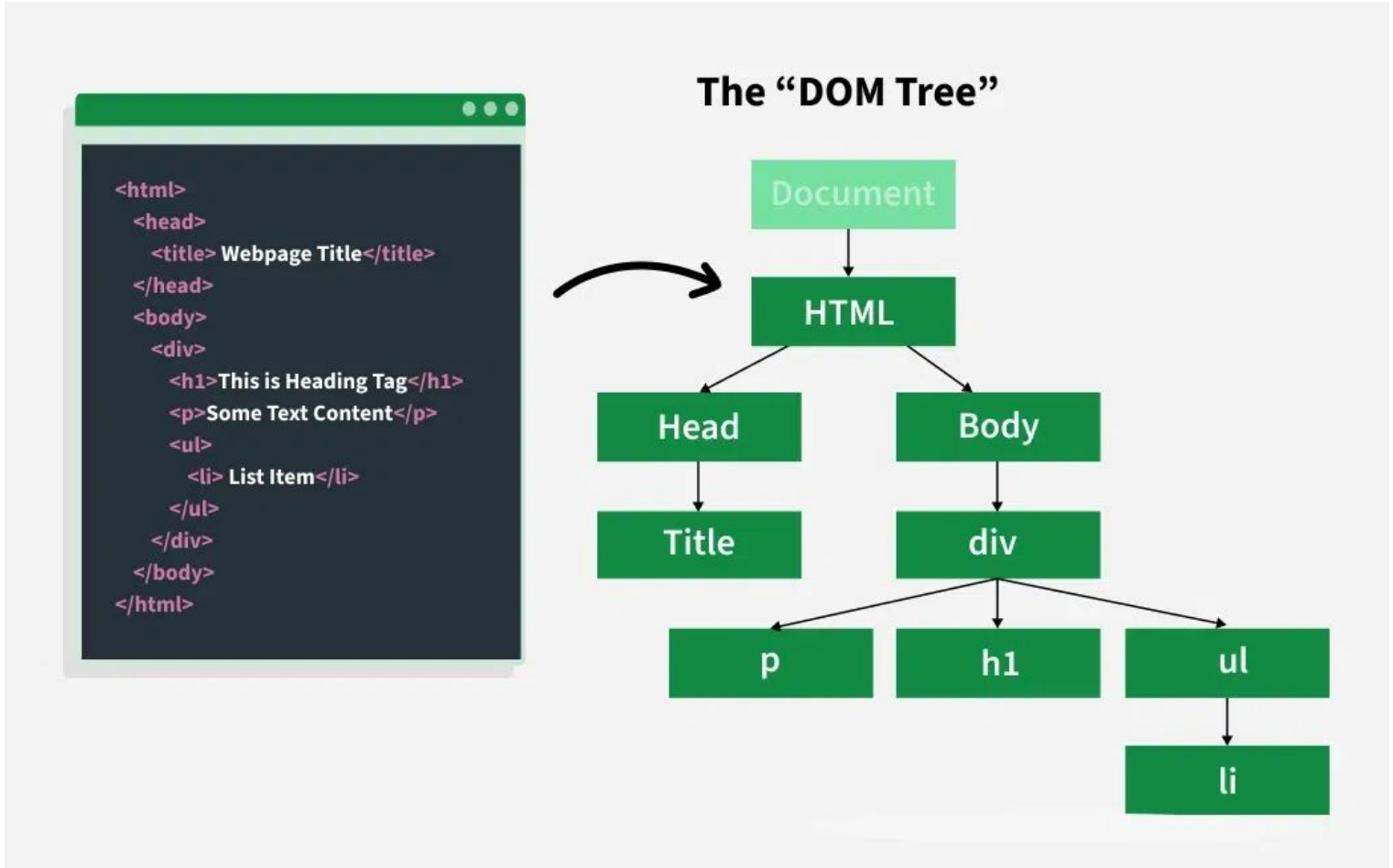
Simplified DOM tree for an example HTML document:

Sample HTML:

```html
<!DOCTYPE html>
<html>
<head>
    <title>DOM Example</title>
</head>
<body>
    <h1>Hello, World!</h1>
    <p>This is a paragraph.</p>
</body>
</html>
```

Corresponding DOM Tree:




The “DOM Tree”
<html>
<head>
<title> Webpage Title</title>
</head>
<body>

<h1>This is Heading Tag</h1>
Some Text Content

 List Item

</body>
</html>
Document
HTML
Head
Body
Title
div
p
h1
ul
li

Types of Nodes in DOM

1. **Element Nodes**:
   - Represent HTML elements like `<div>`, `<h1>`, `<p>`.
   - Example: `<p>This is a paragraph.</p>` is represented as an element node.
2. **Text Nodes**:
   - Contain the actual text inside elements.
   - Example: "This is a paragraph." inside `<p>`.
3. **Attribute Nodes**:
   - Represent attributes of elements.
   - Example: class="example" in `<div class="example">`.
4. **Document Node**:
   - The root node that represents the entire document.

DOM Properties and Methods

Accessing Nodes:

Property/Method	Description	Example
document.getElementById()	Selects an element by its **id** attribute.	document.getElementById('header')
document.querySelector()	Selects the first element matching a CSS selector.	document.querySelector('.class')
document.querySelectorAll()	Selects all elements matching a CSS selector.	document.querySelectorAll('div')
.children	Returns child elements of a node.	document.body.children

DOM Properties and Methods Contd.

Manipulating Nodes:

Property/Method	Description	Example
.innerHTML	Sets/retrieves HTML inside an element.	element.innerHTML = 'Bold'
.textContent	Sets/retrieves text inside an element.	element.textContent = 'Hello'
.style	Modifies CSS styles directly.	element.style.color = 'red'
.setAttribute()	Adds or modifies an attribute.	element.setAttribute('class', 'red')
.appendChild()	Adds a new child node.	parent.appendChild(childNode)

Locator Strategies in Selenium

Locators are a crucial part of Selenium automation, as they allow you to identify web elements on a webpage. Understanding and effectively using locator strategies is key to writing reliable and maintainable test scripts.

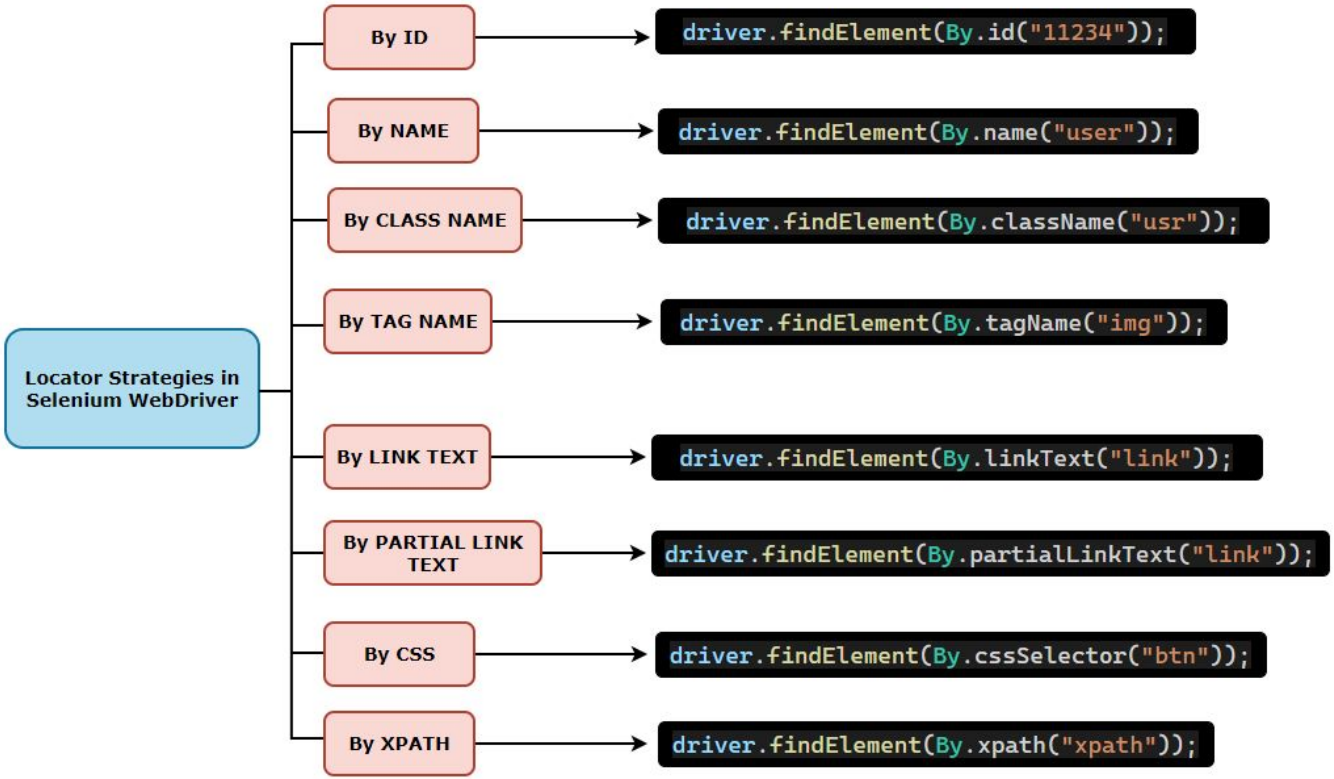
Definition:

- Locators are **identifiers** used in Selenium to find and interact with elements on a webpage.
- They help automate tasks like clicking buttons, filling forms, or verifying content.

Importance:

- Accurate locators ensure tests are reliable and resilient to changes.
- Poorly chosen locators can lead to flaky tests.

Locator Strategies in Selenium



Locator Strategies in Selenium Contd.

Locator Strategy	Advantages	Disadvantages	Best Use Case
ID *driver.findElement(By.id("username")) .sendKeys("testuser");*	➢ Fastest and most efficient. ➢ Unique if properly implemented.	➢ May not always be available for all elements. ➢ If auto-generated, id values can change dynamically.	➢ Use when the id attribute is unique and stable across the application.
Name *driver.findElement(By.name("password")).send Keys("password123");*	➢ Easy to use and read. ➢ Works well for forms and input fields.	➢ May not be unique, leading to conflicts. ➢ Not always available or meaningful.	➢ Use for forms where input fields often have meaningful name attributes.
Class Name *driver.findElement(By.className("login-button ")).click();*	➢ Useful for identifying groups of elements. ➢ Helps when class indicates functionality (e.g., buttons, alerts).	➢ May not be unique, especially in styled elements. ➢ Can match multiple elements, leading to ambiguity	➢ Use for styling-specific elements or identifying groups like error messages or buttons.
Tag Name *List<WebElement> rows = driver.findElements(By.tagName("tr"));*	➢ Ideal for grouping similar elements like tables, lists, or forms. ➢ Simplifies actions on specific types of elements.	➢ Too generic; often matches multiple elements. ➢ Cannot distinguish between elements of the same type.	➢ Use for bulk operations (e.g., iterating over all rows in a table).

Locator Strategies in Selenium Contd.

Locator Strategy	Advantages	Disadvantages	Best Use Case
Link Text *driver.findElement(By.linkText("Forgot Password?")).click();*	➢ Works well for hyperlinks with descriptive text. ➢ Easy to implement for static content.	➢ Fails if the link text is dynamic or partially hidden. ➢ Breaks if the text changes due to localization or dynamic updates.	➢ Use when interacting with static hyperlinks with descriptive and stable text.
Partial Link Text *driver.findElement(By.partialLinkText("Forgot")).click();*	➢ Useful for links with lengthy or partially known text. ➢ Flexible in dynamic content scenarios.	➢ Prone to errors if multiple links contain the same partial text. ➢ Slower compared to other locators as it checks substrings.	➢ Use for lengthy links or where only part of the text is known.
CSS Selector *driver.findElement(By.cssSelector("input[type= 'text']")).sendKeys("username");*	➢ Highly flexible and efficient. ➢ Supports locating elements by hierarchy, attributes, pseudo-classes. ➢ Faster than XPath in most browsers.	➢ Requires knowledge of CSS syntax. ➢ Can be complex to write and debug for nested elements.	➢ Use for elements with dynamic or complex attributes and styles.
XPath *driver.findElement(By.xpath("//div[@id='login-form']//input[@name='username']")) .sendKeys("testuser");*	➢ Extremely powerful and versatile. ➢ Can locate elements based on position, attributes, and relationships. ➢ Supports both absolute (/) and relative (//) paths.	➢ Slower compared to CSS Selectors. ➢ Fragile; absolute XPath breaks with minor changes in DOM structure. ➢ Complex paths are difficult to debug and maintain.	➢ Use for deeply nested or dynamically generated elements where other locators fail.

How to Choose Best Locator Strategy

1. **Start with Simplicity**:
    - Prefer ID and Name for their speed and ease of use.
2. **Handle Styling Scenarios**:
    - Use Class Name and CSS Selector for elements influenced by styles.
3. **Dynamic Content**:
    - Leverage XPath for complex hierarchies and dynamic attributes.
4. **Hyperlinks**:
    - Use Link Text or Partial Link Text for navigation.
5. **Iterative Actions**:
    - Use Tag Name for operations on groups of similar elements like rows or lists.

XPath Locators in Selenium

XPath (XML Path Language) is a powerful and flexible way to locate elements in a web page's DOM structure. It is particularly useful for identifying elements that lack unique attributes such as id or class.

What is XPath?

- XPath is a query language for selecting nodes from an XML document.
- In Selenium, it is used to traverse the DOM to locate web elements.

Why Use XPath?

- Can locate elements with dynamic or complex attributes.
- Supports navigation of parent-child and sibling relationships.

Types of XPath Locators

1. Absolute XPath

- Starts from the root node (**/**) and traverses through each node.
- **Syntax**: **/html/body/div/form/input**

Example:

```
driver.findElement(By.xpath("/html/body/div[2]/form/input[1]")).sendKeys("testuser");
```

2. Relative XPath

- Starts from anywhere in the DOM (**//**) and locates elements dynamically.
- **Syntax**: **//tagname[@attribute='value']**

Example:

```
driver.findElement(By.xpath("//input[@id='username']")).sendKeys("testuser");
```

Types of XPath Locators Contd.

	Absolute XPath	Relative XPath
Description	Absolute XPath starts with the root node or a forward slash (**/**) and follows the full path from the root node to the specific element.	Relative XPath starts with a double forward slash (**//**) and can search for the target element from anywhere in the document.
Example	**/html/body/div[1]/section/div[2]/div**	**//div[@class='example']**
Advantages	1. Precise identification of elements. 2. Not affected by changes in other parts of HTML structure unrelated to the absolute path.	1. Simpler and shorter. 2. More efficient as it doesn't rely on the entire path. 3. Not affected by changes in upper level nodes.
Disadvantages	1. Lengthy and complex. 2. Affected by any structural changes in HTML/XML document leading to automation failures.	1. If not properly used, it may identify more elements. 2. Less precise when compared to absolute xpath.
Best used when	You want to access the element with the exact hierarchy of HTML/XML document.	To identify elements with unique attributes or when the HTML structure changes often, making absolute xpath unreliable.
When to avoid	When the HTML/XML document changes frequently.	When there are multiple elements with similar attributes and it is complex to find the correct one using relative XPath.

XPath Syntax and Patterns

Syntax	Description	Example
//tagname	Finds all elements of the given tag.	//input
//tagname[@attribute='value']	Finds an element with the specified attribute-value pair.	//input[@type='text']
//tagname[text()='value']	Locates elements by visible text.	//button[text()='Submit']
//tagname[contains(@attr,'val')]	Matches elements where the attribute contains the substring.	//div[contains(@class, 'error')]
//tagname[starts-with(@attr,'val')]	Matches elements where the attribute starts with the substring.	//input[starts-with(@id, 'user_')]
//*	Matches all elements in the DOM.	//*

XPath Functions

Function Name	Usage	Example
contains()	Tests if the first argument string contains the second argument string	`` `//h2[contains(text(), 'Intro')]` ``
starts-with()	Tests if the first argument string starts with the second argument string	`` `//a[starts-with(@href, 'https://example')]` ``
text()	Selects all the text children of the context node	`` `//h1[text()='Learning XPath']` ``
not()	Returns true if the argument is false, and false otherwise	`` `//a[not(@href='https://example.org/docs')]` ``
or	Returns true if either of the arguments is true	`` `//a[@href='https://example.com/tutorial' or @href='https://example.org/docs']` ``
and	Returns true only if both the arguments are true	`` `//h2[@class='subheading' and @loc='bottom']` ``
last()	Returns a number equal to the context size from an expression	`` `//p[last()]` ``
position()	Returns the position of the context node in its context	`` `//p[position() = 1]` ``
count()	Returns a number representing the count of a selected number of nodes	`` `count(//p)` ``
string()	Returns the string-value of a specified node	`` `string(//div[@id='main'])` ``
concat()	Concatenates arguments and returns the resulting string	`` `concat(//h1, //p)` ``
normalize-space()	Strips leading and trailing white-space from a string, replaces sequences of whitespace characters by a single space, and returns the resulting string	`` `//p[normalize-space()='First Para in Container']` ``
translate()	Returns a string where some specified characters are replaced with some other specified characters	`` `translate(//p, "abcdef", "ABCDEF")` ``
true()	Returns true	`` `//h2[true()]` ``
false()	Returns false	`` `//div[false()]` ``

XPath Strategy	Example	Description
Absolute XPath	**/html/body/div/p**	Selects the 'p' element under 'div' which is directly under 'body' from the root. This is the complete path from the root element to the desired element.
Relative XPath	**//div/p**	Selects any 'p' element directly under 'div' from anywhere in the document.
XPath Using Single Attribute	**//button[@id='btn-one']**	Selects the 'button' element where attribute 'id' is exactly 'btn-one'.
XPath Using Multiple Attributes	**//a[@href='https://example.com'][@class='external-link']**	Selects the 'a' (link) element where attribute 'href' is 'https://example.com' and 'class' is 'external-link'.
XPath Using Logical Operators **OR**	**//a[@href='https://example.com/tutorial' or @href='https://example.org/docs']**	Selects 'a' elements with 'href' attribute that's either 'https://example.com/tutorial' or 'https://example.org/docs'.
XPath Using Logical Operators **AND**	**//h2[@class='subheading' and @loc='bottom']**	Selects 'h2' elements that have a class 'subheading' and attribute 'loc' with value 'bottom'.

XPath Strategy	Example	Description
XPath Functions contains()	`//h2[contains(text(), 'Intro')]`	Selects 'h2' elements where the text contains the string 'Intro'.
	`//button[contains(@id, 'btn')]`	Selects the 'button' element where attribute 'id' contains the substring 'btn'.
XPath Functions starts-with()	`//a[starts-with(@href, 'https://example')]`	Selects 'a' elements where the 'href' attribute starts with the string 'https://example'.
	`//button[starts-with(@id,  'btn')]`	Selects the 'button' element where attribute 'id' starts with the substring 'btn'.
XPath Functions text()	`//h1[text()='Learning XPath']`	Selects 'h1' elements where the text is exactly 'Learning XPath'.
	`//button[text()='Button One']`	Selects the 'button' element where the text inside the 'button' element is exactly 'Button One'.
XPath Functions last()	`//p[last()]`	Selects the last 'p' element in the document.
XPath Functions position()	`//p[position() = 1]`	Selects the first 'p' element in the document.

XPath Axes

Axes Name	Usage	Example
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node.	`` `//li/ancestor::div` ``
ancestor-or-self	Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself.	`` `//div[@class='links']/ancestor-or-self::div` ``
attribute	Selects all attributes of the current node.	`` `//div[@class='links']/attribute::*` ``
child	Selects all children of the current node.	`` `//div[@id='content']/child::h2` ``
descendant	Selects all descendants (children, grandchildren, etc.) of the current node.	`` `//div[@id='content']/descendant::a` ``
descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself.	`` `//div[@class='links']/descendant-or-self::*` ``
following	Selects everything in the document after the closing tag of the current node.	`` `//h2[@loc='top']/following::*` ``
following-sibling	Selects all siblings after the current node.	`` `//h2[@loc='top']/following-sibling::*` ``
namespace	Selects all namespace nodes of the current node.	The use of namespaces in HTML is rare, so this axis has limited usage in Selenium.
parent	Selects the parent of the current node.	`` `//h2[@loc='top']/parent::div` ``
preceding	Selects everything in the document before the start tag of the current node.	`` `//h2[@loc='bottom']/preceding::*` ``
preceding-sibling	Selects all siblings before the current node.	`` `//h2[@loc='bottom']/preceding-sibling::*` ``
self	Selects the current node.	`` `//div[@class='links']/self::div` ``

XPath Strategy	Example	Description
XPath Axes: Following	**//p[text()='First Para in Container']/following::p**	Selects all 'p' elements in the document that come after (not first child but following sibling) the 'p' element containing the text 'First Para in Container'.
XPath Axes: Preceding	**//p[text()='Second Para in Container']/preceding::p**	Selects all 'p' elements in the document that come before (not the last child but preceding sibling) the 'p' element containing the text 'Second Para in Container'.
XPath Axes: Ancestor	**//li/ancestor::div**	Selects all 'div' ancestors (parent grandparent etc) of the current 'li' element.
XPath Axes: Child	**//div[@id='content']/child::h2**	Selects all 'h2' elements which are children of the 'div' with ID 'content'.
XPath Axes: Descendant	**//div[@id='content']/descendant::a**	Selects any 'a' that is a child (direct or indirect) of the 'div' with ID 'content'.
XPath Axes: Following-sibling	**//h2[@loc='top']/following-sibling::p**	Selects all 'p' siblings that follow the 'h2' with attribute 'loc' 'top'.
XPath Axes: Preceding-sibling	**//h2[@loc='bottom']/preceding-sibling::p**	Selects all 'p' siblings that precede the 'h2' with 'loc' attribute equals 'bottom'.

Best Practices for XPath

1. Prefer Relative XPath

- Avoid fragile absolute paths.
- Use meaningful attributes.

2. Validate XPath

- Use browser DevTools or tools like SelectorsHub to test XPath queries.

3. Combine Functions and Axes

- Leverage functions like contains() and axes for dynamic and complex scenarios.

4. Optimize Performance

- Keep XPath queries concise to improve test execution speed.

5. Avoid Over-reliance on XPath

- Use other locators (e.g., ID or CSS) when simpler options exist.

CSS Selectors in Selenium

CSS Selectors are one of the most efficient and flexible ways to locate elements in Selenium WebDriver. They allow us to target HTML elements based on their attributes, classes, IDs, or other properties defined in the CSS.

What is a CSS Selector?

- CSS Selectors are patterns used to select and style elements in HTML.
- In Selenium, they are used to locate elements in the DOM, offering a concise way to interact with the page.

Why Use CSS Selectors?

- CSS Selectors are generally faster than XPath in most browsers.
- They are easier to read and write for most developers.
- CSS Selectors work well when locating elements based on classes, IDs, and other attributes.

Basic Syntax of CSS Selectors

CSS Selectors are made up of different components:

Selector Type	Description	Example
Element Selector	Selects all elements of a given tag.	div, input, button
ID Selector	Selects an element by its unique ID.	#username
Class Selector	Selects all elements with the specified class.	.form-group
Universal Selector	Selects all elements in the DOM.	*
Attribute Selector	Selects elements with a specific attribute.	[type='text']

CSS Selector Examples

Selector Type	Description	CSS Selector	Example Code
Element Selector	Selects all elements of a given tag.	**button**	driver.findElement(By.cssSelector("button")) .click();
ID Selector	Selects an element by its unique ID.	**#loginButton**	driver.findElement(By.cssSelector("#loginButton")) .click();
Class Selector	Selects all elements with the specified class.	**.login-form**	driver.findElement(By.cssSelector(".login-form")) .submit();
Attribute Selector	Selects elements with a specific attribute.	**[name='username']**	driver.findElement(By.cssSelector("[name='username']")).sendKeys("testuser");
Combining Selectors	Combines selectors to narrow down the selection.	**input[type='password']**	driver.findElement(By.cssSelector("input[type='password']")).sendKeys("1234");
Descendant Selector	Selects elements that are descendants of another element.	**div .login-button**	driver.findElement(By.cssSelector("div .login-button")).click();

CSS Selector Examples Contd.

Selector Type	Description	CSS Selector	Example Code
Child Selector (>)	Selects direct child elements of a given element.	**div > p**	driver.findElement(By.cssSelector("div > p")).click();
Adjacent Sibling (+)	Selects an element immediately preceded by a sibling.	**h2 + p**	driver.findElement(By.cssSelector("h2 + p")).click();
General Sibling (~)	Selects all sibling elements that follow a given element.	**h2 ~ p**	driver.findElement(By.cssSelector("h2 ~ p")).click();
Grouping Selector (,)	Combines multiple selectors into one.	**h1, h2, h3**	driver.findElement(By.cssSelector("h1, h2, h3")).click();

CSS Selectors with Pseudo-Classes

Pseudo-Class Selector	Description	CSS Selector	Example Code
:nth-child(n)	Matches elements based on their position in a parent.	**ul li:nth-child(3)**	driver.findElement(By.cssSelector("ul li:nth-child(3)")).click();
:first-child	Matches the first child element of a parent.	**ul li:first-child**	driver.findElement(By.cssSelector("ul li:first-child")).click();
:last-child	Matches the last child element of a parent.	**ul li:last-child**	driver.findElement(By.cssSelector("ul li:last-child")).click();
:not(selector)	Matches all elements except those specified by selector.	**input:not([type='submit'])**	driver.findElement(By.cssSelector("input:not([type='submit'])")).sendKeys("text");
:hover	Applies to an element when it is hovered over (limited to styles, not Selenium).	**a:hover**	*Note: Cannot be directly used in Selenium but is useful for styles.*

Advantages and Disadvantages of CSS Selectors

Advantages	**Disadvantages**
Faster than XPath in most browsers.	Complex selectors can become difficult to maintain.
Easy to write and read.	Can be less precise than XPath in complex DOM structures.
Supports various combinators and pseudo-classes.	Does not support navigating relationships as robustly as XPath.
More efficient than XPath in browsers like Chrome and Firefox.	May not be as flexible for deeply nested elements.

Best Practices for Using CSS Selectors

1. **Use ID and Class Selectors When Possible**
   - Opt for #id and .class selectors when they are unique to improve performance.
2. **Avoid Complex Selectors**
   - Keep your CSS selectors simple to avoid confusion and maintainability issues.
3. **Use nth-child() and Pseudo-Classes Efficiently**
   - These allow you to target specific elements based on their position or attributes, but avoid overuse.
4. **Validate Selectors**
   - Test your selectors using the browser's developer tools or a tool like SelectorsHub.
5. **Fallback to XPath for Complex Relationships**
   - If CSS Selectors become too cumbersome, fall back on XPath for more complex DOM navigations.

XPath vs CSS Selectors

Description	XPath	CSS Selector
ID	//input[@id='first-name']	input#first-name
Name	//input[@name='user']	input[name='user’]
Classname	//div[@class='container’]	div.container
Text	//a[contains(text(),'Sign in')]	N.A
Multiple Attributes	//input[@id='userName' and @type='text']	input[id='userName'][type='text']
Attribute Contains	//input[contains(@placeholder, 'First')]	input[placeholder*='First']
Attribute Startswith	//label[starts-with(@class,'email')]	label[class^='email']
Child	//div[@name='abcde']/input	div[name='abcde']>input
Parent	//div[@name='pass']/parent::div	N.A
Ancestor	//div[@id='uname']/ancestor::div	N.A
Following-sibling	//div[@name='abcde']/following-sibling::input	div[@name='abcde']+ input
Preceding-sibling	//td[text()='abcde']/preceeding-sibling::input	N.A
nth-child	//div[@name='pass']//input[n]	div[name='pass'] input:nth-child(6)
First Child	//div[@name='pass']//input[1]	div[name='pass'] input:first-child

Relative Locators

- Relative Locators, introduced in Selenium 4, allows us to locate elements based on their position relative to other nearby elements on the web page.
- This feature is particularly useful for identifying elements in dynamic layouts or when unique attributes are unavailable.
- Methods introduced:
    a. above()
    b. below()
    c. toLeftOf()
    d. toRightOf()
    e. near()

Why Use Relative Locators?

- Simplifies automation for dynamic and visually-aligned elements.
- Reduces dependency on brittle or unstable locators.

Relative Locators - Syntax & Example

Basic Syntax:

```
WebElement element = driver.findElement(RelativeLocator.with(By.tagName("tag")).above(referenceElement));
```

Example Usage:

```
// Example: Locate a button above a specific textbox
WebElement textbox = driver.findElement(By.id("username"));
WebElement loginButton = driver.findElement(RelativeLocator.with(By.tagName("button")).above(textbox));
loginButton.click();
```

Relative Locators - Supported Methods

Method	Description	Example Code
above()	Locates an element positioned above a reference element.	**RelativeLocator.with(By.tagName("label")).above(inputBox)**
below()	Locates an element positioned below a reference element.	**RelativeLocator.with(By.tagName("button")).below(headerElement)**
toLeftOf()	Locates an element positioned to the left of a reference element.	**RelativeLocator.with(By.tagName("div")).toLeftOf(buttonElement)**
toRightOf()	Locates an element positioned to the right of a reference element.	**RelativeLocator.with(By.tagName("span")).toRightOf(imageElement)**
near()	Locates an element within 50 pixels (default) or a specified distance.	**RelativeLocator.with(By.tagName("input")).near(icon, 100)**

Relative Locators - Practical Examples

Example 1: Locate an Input Field Below a Label:

```
WebElement label = driver.findElement(By.id("emailLabel"));
WebElement emailField = driver.findElement(RelativeLocator.with(By.tagName("input")).below(label));
emailField.sendKeys("example@test.com");
```

Example 2: Locate a Button to the Right of a Textbox:

```
WebElement usernameField = driver.findElement(By.id("username"));
WebElement loginButton = driver.findElement(RelativeLocator.with(By.tagName("button")).toRightOf(usernameField));
loginButton.click();
```

Example 3: Locate a Checkbox Near a Label:

```
WebElement label = driver.findElement(By.xpath("//label[text()='Remember Me']"));
WebElement checkbox = driver.findElement(RelativeLocator.with(By.tagName("input")).near(label));
checkbox.click();
```

Relative Locators - Benefits & Limitations

Benefits:

1. **Simplified Locators:** Reduces the need for complex XPath or CSS selectors.
2. **Handles Visual Alignments:** Efficiently identifies elements based on their position.
3. **Robust Against DOM Changes:** Less likely to break due to minor changes in the HTML structure.
4. **Improves Readability:** Makes test scripts more intuitive and easier to understand.

Limitations:

1. **Dependency on Nearby Elements:** Fails if the reference element is missing or dynamic.
2. **Limited Use Cases:** May not work well for elements without a consistent spatial relationship.
3. **Less Control over Specific Attributes:** Cannot directly reference specific attributes or hierarchical levels.

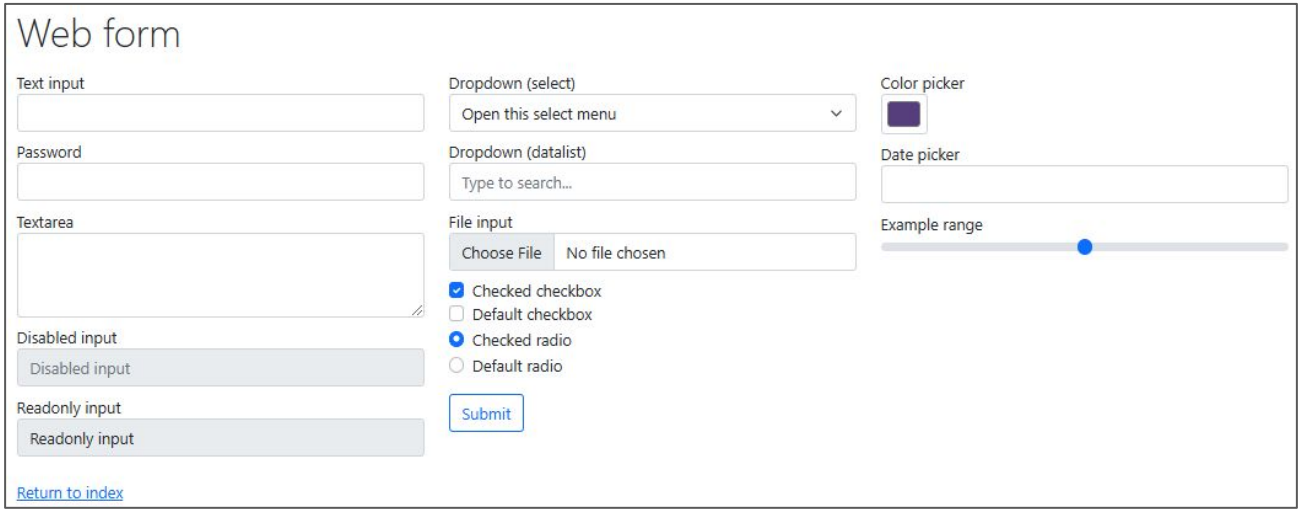
Handling Different Web Controls with WebDriver

What are Web Controls?

- Web controls are interactive elements on a webpage that users can interact with to perform specific actions.
- Examples: Textboxes, buttons, checkboxes, dropdown menus, alerts, and more.

Why is it Important?

- To automate functional testing of web applications, understanding how to interact with these controls programmatically is critical.



What is a WebElement?

- A `WebElement` represents an HTML element on a webpage.
- It provides methods to interact with elements, such as clicking buttons, entering text, selecting options, and more.

Key Methods of WebElement:

Method	Purpose	Example
click()	Clicks the element.	button.click();
sendKeys(String text)	Enters text into an input field.	textbox.sendKeys("example");
getText()	Retrieves visible text of the element.	String text = element.getText();
isDisplayed()	Checks if the element is visible on the page.	boolean visible = element.isDisplayed();
isEnabled()	Checks if the element is enabled for interaction.	boolean enabled = element.isEnabled();
isSelected()	Checks if a checkbox or radio button is selected.	boolean selected = checkbox.isSelected();

Interacting with Textboxes

How to Handle Textboxes

- Use **sendKeys()** to input text into a textbox.
- Use **getDomAttribute("value")** to retrieve the entered text.

Example:

```
WebElement username = driver.findElement(By.id("username"));
username.sendKeys("testuser");  // Enter text
String enteredText = username.getDomAttribute("value");  // Retrieve text
```

Handling Buttons & Links

How to Handle Buttons and Links

- Use **click()** to simulate a button/link click.

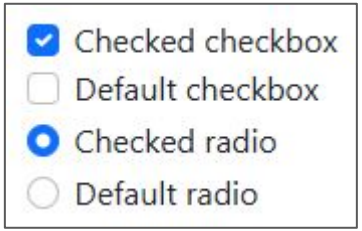
Example:

```
WebElement loginButton = driver.findElement(By.id("loginButton"));
loginButton.click();  // Click the button
```

Working with Checkboxes & Radio buttons

How to Handle Checkboxes

- Use **click()** to check/uncheck.
- Use **isSelected()** to verify the checkbox state.



Example:

```
WebElement rememberMe = driver.findElement(By.id("rememberMe"));
rememberMe.click();  // Check or uncheck the checkbox
boolean isChecked = rememberMe.isSelected();  // Verify if the checkbox is checked
```

How to Handle Radio Buttons

- Use **click()** to select/unselect a radio button.
- Use **isSelected()** to verify the selection state.

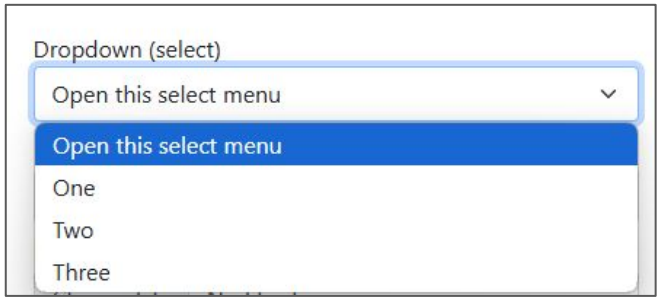
Example:

```
WebElement genderMale = driver.findElement(By.id("genderMale"));
genderMale.click();  // Select the radio button
boolean isMaleSelected = genderMale.isSelected();  // Check if it's selected
```

Working with Dropdowns

How to Handle Dropdowns

- Use the **Select** class to interact with dropdown menus.
- Methods in **Select** class:
  - selectByVisibleText(String text)
  - selectByIndex(int index)
  - selectByValue(String value)



Example:

```
Select countryDropdown = new Select(driver.findElement(By.id("country")));
countryDropdown.selectByVisibleText("India");  // Select by visible text
```

Commonly Used Methods on a WebElement

Method	Returns	Purpose	Example
click()	void	Simulates a mouse click on the element.	button.click();
sendKeys(String text)	void	Types the provided text into the element, typically an input field.	textbox.sendKeys("Hello World");
clear()	void	Clears the content of an input or textarea element.	textbox.clear();
getText()	String	Retrieves visible text of the element.	String text = element.getText();
getCssValue(String name)	String	Retrieves the computed value of a CSS property.	String color = element.getCssValue("color");
getTagName()	String	Returns the tag name of the element (e.g., div, input).	String tagName = element.getTagName();
isDisplayed()	boolean	Checks if the element is visible on the page.	boolean visible = element.isDisplayed();
isEnabled()	boolean	Checks whether the element is enabled and can be interacted with.	boolean enabled = element.isEnabled();
isSelected()	boolean	Checks if a checkbox or radio button is selected.	boolean selected = checkbox.isSelected();

Commonly Used Methods on a WebElement Contd.

Method	Returns	Purpose	Example
getSize()	Dimension	Retrieves the width and height of the element as a Dimension object.	Dimension size = element.getSize();
getLocation()	Point	Retrieves the location (x, y coordinates) of the element on the page.	Point location = element.getLocation();
findElement(By locator)	WebElement	Finds a single child element inside the current element.	WebElement child = parent.findElement(By.tagName("span"));
findElements(By locator)	List<WebElement>	Finds all child elements matching the locator inside the current element.	List<WebElement> items = list.findElements(By.tagName("li"));
getRect()	Rectangle	Returns the dimensions and coordinates of the element as a Rectangle object.	Rectangle rect = element.getRect();
submit()	void	Submits a form if the element is inside a <form> tag.	input.submit();

Advanced Interactions with WebDriver

Advanced interactions refer to user actions that involve:

- Mouse and keyboard operations.
- Handling multi-touch or drag-and-drop actions.

Why Advanced Interactions?

- Required for complex test scenarios such as:
  - Hovering over elements.
  - Performing drag-and-drop.
  - Keyboard shortcuts.
  - Multi-element selection.

What is the Actions Class?

- A utility in Selenium for simulating user interactions.
- Part of the org.openqa.selenium.interactions package.

Key Features

- Handles complex mouse and keyboard events.
- Supports actions like hovering, right-clicking, and drag-and-drop.

Basic Mouse Interactions

Action	Method	Example
Hover	**moveToElement(WebElement)**	actions.moveToElement(menu).perform();
Right Click	**contextClick(WebElement)**	actions.contextClick(button).perform();
Double Click	**doubleClick(WebElement)**	actions.doubleClick(element).perform();
Click and Hold	**clickAndHold(WebElement)**	actions.clickAndHold(slider).perform();
Release	**release()**	actions.clickAndHold(slider).release().perform();

Drag-and-Drop

How to Perform Drag-and-Drop

- Use the dragAndDrop() or clickAndHold() + moveToElement() methods.

Example:

```
WebElement source = driver.findElement(By.id("source"));
WebElement target = driver.findElement(By.id("target"));
actions.dragAndDrop(source, target).perform();
```

Alternate Example:

```
actions.clickAndHold(source).moveToElement(target).release().perform();
```

Keyboard Interactions

Action	Method	Example
Send Keys	**sendKeys(String keys)**	actions.sendKeys(Keys.ENTER).perform();
Key Down	**keyDown(Keys key)**	actions.keyDown(Keys.SHIFT).perform();
Key Up	**keyUp(Keys key)**	actions.keyDown(Keys.SHIFT).keyUp(Keys.SHIFT).perform();
Combination	**sendKeys(WebElement, String keys)**	actions.sendKeys(inputField, "text").perform();

Performing Chained Actions

- The Actions class allows chaining of multiple actions in a single statement.

```
actions.moveToElement(menu).click(subMenu).keyDown(Keys.SHIFT).sendKeys("text").perform();
```

Handling Sliders & Resizable Elements

How to Handle Sliders

- Drag the slider handle using clickAndHold() and moveByOffset().

Example:

```
WebElement slider = driver.findElement(By.id("slider"));
actions.clickAndHold(slider).moveByOffset(50, 0).release().perform();
```

How to Handle Resizable Elements

- Use clickAndHold() and moveByOffset() to resize elements.

Example:

```
WebElement resizeHandle = driver.findElement(By.className("ui-resizable-handle"));
actions.clickAndHold(resizeHandle).moveByOffset(100, 50).release().perform();
```

Handling Multi-Element Selection

How to Select Multiple Elements

- Use keyDown(Keys.CONTROL) or keyDown(Keys.SHIFT) to select multiple elements.

Example:

```
List<WebElement> items = driver.findElements(By.className("list-item"));
actions.keyDown(Keys.CONTROL)
       .click(items.get(0))
       .click(items.get(1))
       .click(items.get(2))
       .keyUp(Keys.CONTROL)
       .perform();
```

Commonly Used Methods in the Actions Class

Method	Description	Usage Example
click()	Simulates a single mouse click on the current mouse position or a specified element.	actions.click(element).perform();
clickAndHold()	Simulates a click-and-hold action on the current position or a specified element.	actions.clickAndHold(element).perform();
release()	Releases the click-and-hold action at the current position or a specified element.	actions.clickAndHold(element).release().perform();
doubleClick()	Simulates a double-click action on the current mouse position or a specified element.	actions.doubleClick(element).perform();
contextClick()	Simulates a right-click (context menu) on the current mouse position or a specified element.	actions.contextClick(element).perform();
moveToElement(WebElement)	Moves the mouse to the center of the specified element.	actions.moveToElement(menu).perform();
moveByOffset(int x, int y)	Moves the mouse to a specific offset from the current position.	actions.moveByOffset(50, 100).perform();
dragAndDrop(WebElement, WebElement)	Drags an element from a source location to a target location.	actions.dragAndDrop(source, target).perform();

Commonly Used Methods in the Actions Class Contd.

Method	Description	Usage Example
dragAndDropBy(WebElement, int x, int y)	Drags an element from its current position to an offset position.	actions.dragAndDropBy(element, 100, 50).perform();
keyDown(Keys key)	Simulates pressing a key on the keyboard (e.g., SHIFT, CTRL).	actions.keyDown(Keys.SHIFT).perform();
keyUp(Keys key)	Simulates releasing a pressed key.	actions.keyDown(Keys.SHIFT).keyUp(Keys.SHIFT).perform();
sendKeys(String keys)	Sends a sequence of keys to the current active element or a specified element.	actions.sendKeys(Keys.ENTER).perform();
sendKeys(WebElement, String keys)	Sends a sequence of keys to a specified element	actions.sendKeys(inputField, "test").perform();
pause(Duration duration)	Pauses the action sequence for a specified duration.	actions.pause(Duration.ofSeconds(2)).perform();
build()	Compiles a set of actions into a single, executable action sequence.	actions.moveToElement(menu).click().build().perform();
perform()	Executes all actions in the built sequence.	actions.moveToElement(menu).click().perform();

Handling Alerts

What Are Alerts?

Alerts are **small pop-up windows** generated by the browser to display important information or gather user input. These are part of the browser's native functionality and not HTML DOM elements, which makes handling them a unique process in Selenium WebDriver.

Types of Alerts

1. **Simple Alert**
    - Displays a message with an **OK** button.
    - Example: "Task Completed Successfully!"
2. **Confirmation Alert**
    - Displays a message with **OK** and **Cancel** buttons.
    - Example: "Do you want to delete this file?"
3. **Prompt Alert**
    - Displays a message with an input field for the user to enter text, along with **OK** and **Cancel** buttons.
    - Example: "Enter your name."

WebDriver API for Alerts

Selenium provides the Alert interface under the switchTo().alert() method to handle alerts. Key methods include:

Method	Description
accept()	Clicks the OK button on the alert.
dismiss()	Clicks the Cancel button on the alert.
getText()	Retrieves the message text displayed in the alert.
sendKeys(String text)	Inputs text into a prompt alert's text field.

Examples of Handling Alerts

1. Handling a Simple Alert

Use Case: Clicking a button triggers a success message.

```
// Trigger the alert
driver.findElement(By.id("showAlertButton")).click();

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

// Retrieve and print alert text
System.out.println("Alert Text: " + alert.getText());

// Accept the alert (click OK)
alert.accept();
```

2. Handling a Confirmation Alert

Use Case: Confirming or cancelling an action.

```
// Trigger the confirmation alert
driver.findElement(By.id("deleteButton")).click();

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

// Retrieve and print alert text
System.out.println("Alert Text: " + confirmAlert.getText());

// Dismiss the alert (click Cancel)
confirmAlert.dismiss();
```

Examples of Handling Alerts Contd.

3. Handling a Prompt Alert

Use Case: Entering data into an alert's input field.

```
// Trigger the prompt alert
driver.findElement(By.id("promptButton")).click();

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

// Retrieve and print alert text
System.out.println("Alert Text: " + promptAlert.getText());

// Enter text into the prompt alert
promptAlert.sendKeys("Selenium Test");

// Accept the alert (click OK)
promptAlert.accept();
```

Best Practices for Handling Alerts

1. **Wait for Alerts**: Alerts may take some time to appear. Use explicit waits for better stability.

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
Alert alert = wait.until(ExpectedConditions.alertIsPresent());
alert.accept();
```

2. **Handle Exceptions**: Always anticipate scenarios where alerts may not appear.

```
try {
    Alert alert = driver.switchTo().alert();
    alert.accept();
} catch (NoAlertPresentException e) {
    System.out.println("No alert found.");
}
```

3. **Validate Alert Text**: Always verify the text in the alert to ensure you're interacting with the correct alert.

```
if (alert.getText().equals("Expected Text")) {
    alert.accept();
} else {
    alert.dismiss();
}
```

Handling Windows & Tabs

Understanding Tabs and Windows

1. **Tabs**: Modern browsers often use tabs for navigation within a single browser window.
2. **Windows**: Separate browser windows are used for different browser instances or pop-ups.

Both are handled using unique **window handles**, which can be retrieved using getWindowHandle() (current handle) and getWindowHandles() (all handles).

Key WebDriver Methods for Tabs and Windows

Method	Description
getWindowHandle()	Retrieves the handle of the current browser window or tab.
getWindowHandles()	Retrieves a Set of all open browser window or tab handles.
switchTo().window(String handle)	Switches focus to a specific window or tab using its handle.
close()	Closes the current tab or window.
switchTo().newWindow(WindowType)	Opens a new tab or window based on the WindowType enum (WindowType.TAB or WindowType.WINDOW).

Handling Windows & Tabs Contd.

Example 1: Switching Between Tabs

Scenario

1. Open a new tab.
2. Switch to the new tab, perform an action, and return to the main tab.

```java
// Open the main page
driver.get("https://example.com");

// Get the current window handle
String mainTab = driver.getWindowHandle();

// Open a new tab
WebDriver newTab = driver.switchTo().newWindow(WindowType.TAB);

// Navigate to a different URL in the new tab
newTab.get("https://another-example.com");

// Perform operations in the new tab
System.out.println("New Tab Title: " + driver.getTitle());

// Switch back to the main tab
driver.switchTo().window(mainTab);

// Perform operations in the main tab
System.out.println("Main Tab Title: " + driver.getTitle());
```

Handling Windows & Tabs Contd.

Example 2: Switching Between Windows

Scenario

1. Open a new browser window.
2. Switch to the new window, perform an action, and return to the main window.

```
// Open the main page
driver.get("https://example.com");

// Get the current window handle
String mainWindow = driver.getWindowHandle();

// Open a new window
WebDriver newWindow = driver.switchTo().newWindow(WindowType.WINDOW);

// Navigate to a different URL in the new window
newWindow.get("https://another-example.com");

// Perform operations in the new window
System.out.println("New Window Title: " + driver.getTitle());

// Close the new window
newWindow.close();

// Switch back to the main window
driver.switchTo().window(mainWindow);

// Perform operations in the main window
System.out.println("Main Window Title: " + driver.getTitle());
```

Handling Windows & Tabs Contd.

Example 3: Switching Between Multiple Tabs or Windows

Scenario:

Handle multiple tabs/windows using getWindowHandles().

```java
// Open the first page
driver.get("https://example1.com");

// Open the second tab
WebDriver newTab = driver.switchTo().newWindow(WindowType.TAB);
newTab.get("https://example2.com");

// Open the third tab
WebDriver thirdTab = driver.switchTo().newWindow(WindowType.TAB);
thirdTab.get("https://example3.com");

// Retrieve all window handles
Set<String> handles = driver.getWindowHandles();

// Switch between tabs/windows using their handles
for (String handle : handles) {
    driver.switchTo().window(handle);
    System.out.println("Current Tab/Window Title: " + driver.getTitle());
}

// Return to the main tab
String mainTab = handles.iterator().next(); // Assuming the first tab is the main tab
driver.switchTo().window(mainTab);
```

Handling Windows & Tabs Contd.

Example 4: Handle New Tab Opened by a Button Click

```java
// Step 1: Open the main page
driver.get("https://example.com");

// Step 2: Get the current window handle (main tab)
String mainTabHandle = driver.getWindowHandle();

// Step 3: Locate and click the button that opens a new tab
driver.findElement(By.id("openNewTabButton")).click();

// Step 4: Wait for the new tab to open (optional, ensures stability)
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
wait.until(ExpectedConditions.numberOfWindowsToBe(2));

// Step 5: Get all window handles
Set<String> allHandles = driver.getWindowHandles();

// Step 6: Switch to the new tab
for (String handle : allHandles) {
    if (!handle.equals(mainTabHandle)) {
        driver.switchTo().window(handle);
        break;
    }
}

// Step 7: Perform actions in the new tab
System.out.println("New Tab Title: " + driver.getTitle());
driver.findElement(By.id("someElement")).click(); // Example action

// Step 8: Close the new tab
driver.close();

// Step 9: Switch back to the original tab
driver.switchTo().window(mainTabHandle);

// Step 10: Continue actions in the main tab
System.out.println("Main Tab Title: " + driver.getTitle());
```

Best Practices for Tabs and Windows

1. **Keep Track of Handles**: Store the main window/tab handle to easily switch back after operations.
2. **Use Explicit Waits**: Ensure the new tab or window has fully loaded before interacting with elements.
3. **Close Tabs/Windows When Done**: Always close unused tabs/windows to avoid resource leakage.
4. **Validate Titles or URLs**: Ensure correct tab/window switching by validating page titles or URLs.

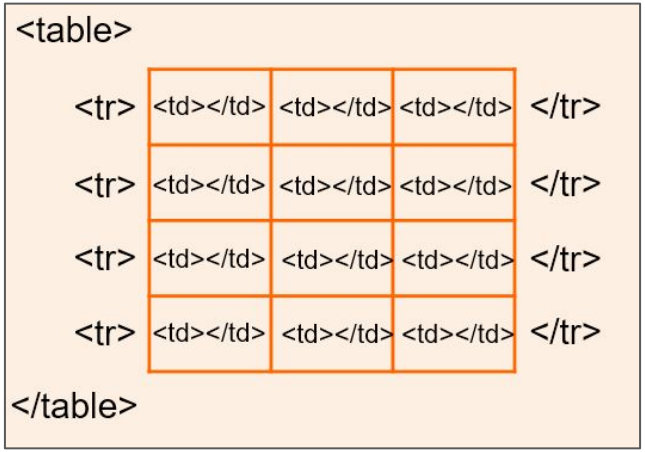
Handling Web Tables

Web tables are commonly used in web applications to display structured data. Interacting with these tables using Selenium WebDriver is an essential skill for test automation.

What Are Web Tables?

A **web table** is an HTML structure created using the <table> tag, consisting of:

- **Rows** (<tr>): Representing a single data record.
- **Columns** (<th> or <td>): Representing attributes or data fields.



Common Scenarios with Web Tables

1. Retrieve and print data from a web table.
2. Find the total number of rows or columns.
3. Extract specific cell data (e.g., the second column of the third row).
4. Search for a value in a table and perform actions on it (e.g., click a button in the same row).

HTML Example: Web Table

```html
<table id="employeeTable">
    <thead>
        <tr>
            <th>Employee ID</th>
            <th>Name</th>
            <th>Department</th>
            <th>Actions</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>101</td>
            <td>Alice</td>
            <td>HR</td>
            <td><button class="edit">Edit</button></td>
        </tr>
        <tr>
            <td>102</td>
            <td>Bob</td>
            <td>Finance</td>
            <td><button class="edit">Edit</button></td>
        </tr>
        <tr>
            <td>103</td>
            <td>Charlie</td>
            <td>IT</td>
            <td><button class="edit">Edit</button></td>
        </tr>
    </tbody>
</table>
```

Examples for Web Table Interactions

1. Retrieve All Table Data

```java
// Locate the table
WebElement table = driver.findElement(By.id("employeeTable"));

// Get all rows in the table body
List<WebElement> rows =
table.findElements(By.xpath("//tbody/tr"));

// Iterate through rows and print data
for (WebElement row : rows) {
    List<WebElement> cells = row.findElements(By.tagName("td"));
    for (WebElement cell : cells) {
        System.out.print(cell.getText() + " ");
    }
    System.out.println();
}
```

2. Find Total Rows and Columns

```java
// Locate the table
WebElement table = driver.findElement(By.id("employeeTable"));

// Count rows in the table body
int rowCount =
table.findElements(By.xpath("//tbody/tr")).size();
System.out.println("Total Rows: " + rowCount);

// Count columns in the header
int columnCount =
table.findElements(By.xpath("//thead/tr/th")).size();
System.out.println("Total Columns: " + columnCount);
```

Examples for Web Table Interactions Contd.

3. Extract Specific Cell Data

Scenario: Retrieve the name of the employee in the second row.

```
// Locate the specific cell (second row, second column)
WebElement cell = driver.findElement(By.xpath("//tbody/tr[2]/td[2]"));
System.out.println("Employee Name: " + cell.getText());
```

4. Perform Actions on Specific Row

Scenario: Click the "Edit" button for the employee with ID 102.

```
// Locate the row containing the specific Employee ID
WebElement row = driver.findElement(By.xpath("//tbody/tr[td[text()='102']]"));

// Click the Edit button in the row
row.findElement(By.className("edit")).click();
```

5. Search for a Value and Retrieve Its Row

Scenario: Find the department of employee Charlie.

```
// Locate the row containing the specific name
WebElement row = driver.findElement(By.xpath("//tbody/tr[td[text()='Charlie']]"));

// Retrieve the department value (third column)
String department = row.findElement(By.xpath("td[3]")).getText();
System.out.println("Charlie's Department: " + department);
```

Best Practices for Web Table Interactions

1. **Dynamic Locators**: Use dynamic XPath to handle tables with changing data or row orders.

```
// Example for dynamic row and column
driver.findElement(By.xpath("//tbody/tr[" + rowIndex + "]/td[" + colIndex + "]"));
```

2. **Avoid Hardcoding**: Write reusable methods for handling web tables dynamically.

```
public String getCellValue(WebElement table, int row, int column) {
    return table.findElement(By.xpath("//tbody/tr[" + row + "]/td[" + column + "]")).getText();
}
```

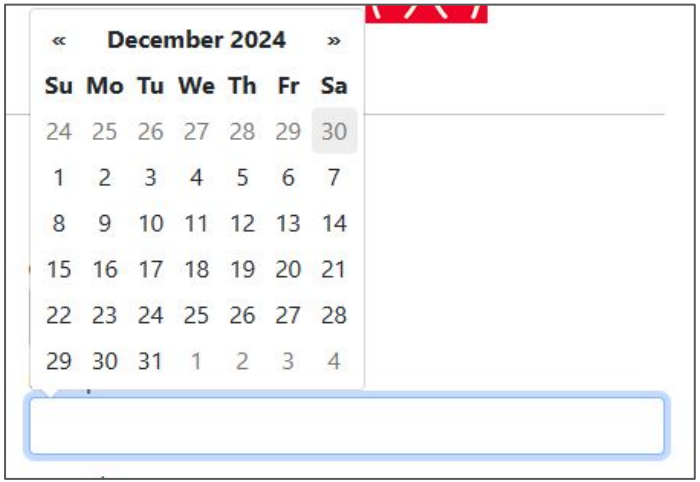
3. **Explicit Waits**: Ensure the table has loaded before performing operations.
4. **Pagination Handling**: If the table spans multiple pages, write logic to navigate through all pages and collect data.

Handling Date Pickers in Selenium WebDriver

Date pickers are a common web control that often mimic table structures, with selectable dates represented as rows and columns. Automating interactions with date pickers requires dynamic element locators and an understanding of the underlying HTML structure.

Common Scenarios with Date Pickers

1. Select a specific date (e.g., "15th May 2024").
2. Select today's date dynamically.
3. Navigate between months or years in the picker.



Wait Mechanisms in Selenium WebDriver

- Synchronization in Selenium WebDriver ensures that the automation script operates at the right pace, waiting for the necessary conditions to be met before proceeding.
- Effective wait mechanisms prevent race conditions and flaky tests by addressing dynamic content loading, slow network responses, or AJAX calls.

Types of Waits in Selenium:

Wait Type	**Description**
Implicit Wait	Sets a global timeout for locating elements. WebDriver waits for the specified time if an element is not found immediately.
Explicit Wait	Waits for a specific condition to be met before proceeding.
Fluent Wait	A more customizable version of Explicit Wait, with polling intervals and exception handling.

Thread.sleep() (Not Recommended) from Java:

Halts the script execution for a fixed duration, irrespective of the condition. Used only in debugging.

Wait Mechanisms - Implicit Wait

Implicit Wait defines a default wait time for all WebDriver operations, applied globally to all elements.

Syntax:

```
WebDriver driver = new ChromeDriver();
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
```

Advantages:

- Simple to use.
- Reduces the need for explicit wait in simpler scenarios.

Disadvantages:

- Applies globally, which may lead to inefficiencies if different elements require different wait times.

Wait Mechanisms - Explicit Wait

Explicit Wait is more precise, allowing WebDriver to wait for specific conditions or elements before proceeding.

Syntax:

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("dynamicElement")));
element.click();
```

Commonly Used Expected Conditions:

Condition	Description
visibilityOfElementLocated	Waits for the element to be visible.
elementToBeClickable	Waits until the element is clickable.
presenceOfElementLocated	Waits for the presence of the element in the DOM.
textToBePresentInElement	Waits for a specific text to be present in an element.
alertIsPresent	Waits for an alert to appear.
elementToBeSelected	Waits until an element is selected.

Advantages:

- Highly specific and precise.
- Targets individual conditions or elements.

Disadvantages:

- Requires more code for implementation.
- May need additional imports (e.g., ExpectedConditions).

Wait Mechanisms - Fluent Wait

Fluent Wait is an advanced form of Explicit Wait, allowing more granular control with polling intervals and exception handling.

Syntax:

```
Wait<WebDriver> wait = new FluentWait<>(driver)
    .withTimeout(Duration.ofSeconds(20))
    .pollingEvery(Duration.ofSeconds(2))
    .ignoring(NoSuchElementException.class);

WebElement element = wait.until(driver -> driver.findElement(By.id("dynamicElement")));
element.click();
```

Advantages:

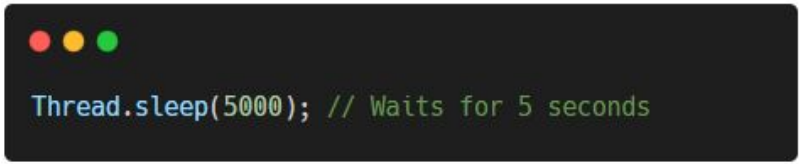
- Customizable polling intervals.
- Handles specific exceptions gracefully.

Disadvantages:

- Slightly more complex to implement.

Wait Mechanisms - Thread.sleep()

Syntax:



Advantages:

- Easy to implement.

Disadvantages:

- Inefficient and unreliable for dynamic web pages.
- May lead to unnecessarily long wait times.

Best Practices for Waits

1. **Prefer Explicit and Fluent Waits**: Use Explicit or Fluent Waits for flexibility and precise control.
2. **Avoid Thread.sleep()**: Use only as a last resort or for debugging purposes.
3. **Use Implicit Wait for Simple Scenarios**: If all elements on the page load within a consistent time, an Implicit Wait can simplify the script.
4. **Set Appropriate Timeouts**: Avoid setting excessively high or low timeout values.

Wait Mechanisms - Examples

1. Waiting for AJAX Content

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(15));
wait.until(ExpectedConditions.invisibilityOfElementLocated(By.id("loadingSpinner")));
WebElement content = driver.findElement(By.id("dynamicContent"));
System.out.println(content.getText());
```

2. Waiting for Alerts

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
wait.until(ExpectedConditions.alertIsPresent());
driver.switchTo().alert().accept();
```

3. Polling for Specific State (Fluent Wait)

```
Wait<WebDriver> wait = new FluentWait<>(driver)
    .withTimeout(Duration.ofSeconds(30))
    .pollingEvery(Duration.ofSeconds(5))
    .ignoring(NoSuchElementException.class);

WebElement button = wait.until(driver -> driver.findElement(By.id("dynamicButton")));
button.click();
```

Comparison of Wait Mechanisms

Wait Type	Use Case	Pros	Cons
Implicit Wait	Static or predictable elements	Simple to implement	Global application may cause delays
Explicit Wait	Dynamic or condition-based elements	Targeted and flexible	Requires additional code
Fluent Wait	Custom polling or complex conditions	Highly customizable	Slightly more complex
Thread.sleep()	Debugging or fixed delays	Easy to use	Inefficient and unreliable

Handling IFrames in Selenium WebDriver

IFrames (Inline Frames) are HTML documents embedded within another HTML document. They are often used for embedding external content, such as ads, videos, or widgets, in web pages. Selenium WebDriver requires special handling to interact with elements inside IFrames.

What is an IFrame?

An IFrame is defined using the <iframe> tag. It can have its own HTML content and acts as an independent browser window embedded within the parent page.

HTML Example of an IFrame:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Course - Real-Time Iframe</title>
</head>
<body>
    <h1>Welcome to My Course</h1>

    <iframe src="https://www.tube.com/embed/dQw4w9WgXcQ" title="Course Video"></iframe>

    <footer>
        <p>Course content provided by Your Name | &copy; 2024</p>
    </footer>
</body>
</html>
```

Challenges with IFrames

1. Selenium WebDriver cannot directly interact with elements inside an IFrame without switching to it.
2. You need to explicitly switch to the IFrame context before interacting with its elements.
3. Returning to the parent frame after completing actions inside the IFrame is essential for continuing other interactions.

Interacting with IFrame

Switching to an IFrame:

Selenium provides three ways to switch to an IFrame:

1. **By Index**:

   ```
   driver.switchTo().frame(0); // Switch to the first iframe on the page
   ```

2. **By Name or ID**:

   ```
   driver.switchTo().frame("iframe1"); // Switch using the iframe's ID or name
   ```

3. **By WebElement**:

   ```
   WebElement iframeElement = driver.findElement(By.id("iframe1"));

   driver.switchTo().frame(iframeElement); // Switch using the WebElement
   ```

Switching back from an IFrame:

1. **To the Parent Frame**::

   ```
   driver.switchTo().parentFrame(); // Switch back to the immediate parent frame
   ```

2. **To the Main Document**:

   ```
   driver.switchTo().defaultContent(); // Switch back to the top-level frame
   ```

IFrames - Code Examples

1. Interacting with Elements Inside an IFrame

Scenario: Enter text into an input field within an IFrame.

```
<iframe id="iframe1">
    <html>
        <body>
            <input id="nameField" type="text">
        </body>
    </html>
</iframe>
```

```
// Switch to the iframe
driver.switchTo().frame("iframe1");

// Interact with elements inside the iframe
WebElement nameField = driver.findElement(By.id("nameField"));
nameField.sendKeys("Selenium WebDriver");

// Switch back to the main page
driver.switchTo().defaultContent();
```

IFrames - Code Examples Contd.

2. Nested IFrames

Scenario: Handle an iframe inside another iframe.

```
<iframe id="outerFrame">
    <html>
        <body>
            <iframe id="innerFrame">
                <html>
                    <body>
                        <button id="clickMe">Click Me</button>
                    </body>
                </html>
            </iframe>
        </body>
    </html>
</iframe>
```

```
// Switch to the outer iframe
driver.switchTo().frame("outerFrame");

// Switch to the inner iframe
driver.switchTo().frame("innerFrame");

// Interact with the button inside the inner iframe
driver.findElement(By.id("clickMe")).click();

// Switch back to the outer iframe
driver.switchTo().parentFrame();

// Switch back to the main page
driver.switchTo().defaultContent();
```

IFrames - Code Examples Contd.

3. Counting Total IFrames on a Page

```
// Get all iframe elements
List<WebElement> iframes = driver.findElements(By.tagName("iframe"));
System.out.println("Total IFrames on the page: " + iframes.size());
```

4. Handling Dynamic IFrames

Dynamic iframes may not have static IDs or names. In such cases, locate them by attributes or index.

```
// Locate iframe by a unique attribute
WebElement dynamicIframe = driver.findElement(By.xpath("//iframe[contains(@src, 'example')]"));
driver.switchTo().frame(dynamicIframe);
```

Best Practices for IFrame Automation

1. **Switch Back After Actions**: Always switch back to the parent frame or main document after interacting with the iframe.
2. **Identify IFrame Uniquely**: Use ID, Name, or other unique attributes to locate the iframe for reliable automation.
3. **Handle Nested IFrames**: Use a step-by-step approach to navigate nested iframes.
4. **Explicit Waits**: Wait for the iframe to load fully before switching.

JavaScriptExecutor in Selenium WebDriver

JavaScript Executor is a powerful feature in Selenium WebDriver that allows direct execution of JavaScript in the context of the browser. It is particularly useful for interacting with elements or functionalities that standard WebDriver methods cannot handle.

Why Use JavaScript Executor?

- To handle elements that are not directly accessible via WebDriver locators.
- To perform advanced actions like scrolling, manipulating attributes, or triggering events.
- To interact with Shadow DOM (if native WebDriver methods are unavailable).
- To retrieve browser or DOM-related information.

How to Use JavaScriptExecutor

JavaScript Executor is an interface provided by Selenium, and it can be used by casting the WebDriver instance to JavascriptExecutor.

Syntax:

```
JavascriptExecutor js = (JavascriptExecutor) driver;
```

Methods:

1. **executeScript(String script, Object... args)**:
   - Executes the provided JavaScript code in the context of the browser.
   - Can accept arguments to interact with specific elements.
2. **executeAsyncScript(String script, Object... args)**:
   - Executes asynchronous JavaScript code.
   - Useful for scenarios like waiting for asynchronous operations to complete.

Common Use Cases of JavaScriptExecutor

1. Scrolling the Page

Scroll to a specific position:

```
JavascriptExecutor js = (JavascriptExecutor) driver;
js.executeScript("window.scrollBy(0, 500);"); // Scroll down by 500 pixels
```

Scroll to a specific element:

```
WebElement element = driver.findElement(By.id("targetElement"));
js.executeScript("arguments[0].scrollIntoView(true);", element);
```

Common Use Cases of JavaScriptExecutor Contd.

2. Interacting with Hidden or Disabled Elements

Click on a hidden element:

```
WebElement hiddenButton = driver.findElement(By.id("hiddenButton"));
js.executeScript("arguments[0].click();", hiddenButton);
```

Enable a disabled element:

```
WebElement disabledInput = driver.findElement(By.id("disabledInput"));
js.executeScript("arguments[0].removeAttribute('disabled');", disabledInput);
```

Common Use Cases of JavaScriptExecutor Contd.

3. Retrieving Element or DOM Properties

Get the value of an input field:

```
WebElement inputField = driver.findElement(By.id("inputField"));
String value = (String) js.executeScript("return arguments[0].value;", inputField);
```

Retrieve the page title:

```
String title = (String) js.executeScript("return document.title;");
```

Get the innertext of an element:

```
WebElement element = driver.findElement(By.id("targetElement"));
String innerText = (String) js.executeScript("return arguments[0].innerText;", element);
```

Common Use Cases of JavaScriptExecutor Contd.

4. Handling Alerts

Trigger a custom alert:

```
js.executeScript("alert('This is a custom alert!');");
```

Close an alert:

```
driver.switchTo().alert().accept();
```

5. Modifying DOM Elements

Change the background color of an element:

```
WebElement element = driver.findElement(By.id("highlightElement"));
js.executeScript("arguments[0].style.backgroundColor = 'yellow';", element);
```

Add a new element to the DOM:

```
js.executeScript("var newDiv = document.createElement('div'); newDiv.innerHTML =
'New Element'; document.body.appendChild(newDiv);");
```

Common Use Cases of JavaScriptExecutor Contd.

7. Executing Asynchronous Scripts

Wait for an AJAX request to complete:

```
js.executeAsyncScript(
    "var callback = arguments[arguments.length - 1];" +
    "var xhr = new XMLHttpRequest();" +
    "xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts', true);" +
    "xhr.onreadystatechange = function() {" +
    "  if (xhr.readyState == 4 && xhr.status == 200) {" +
    "    callback(xhr.responseText);" +
    "  }" +
    "};" +
    "xhr.send();"
);
```

JavaScriptExecutor Advantages & Disadvantages

Advantages:

Advantage	Description
Bypasses WebDriver Limitations	Can handle non-standard scenarios like hidden elements or Shadow DOM interactions.
Direct Browser Interaction	Executes JavaScript directly in the browser for high flexibility.
Advanced Actions	Allows complex actions like dynamic attribute manipulation or custom events.

Disadvantages:

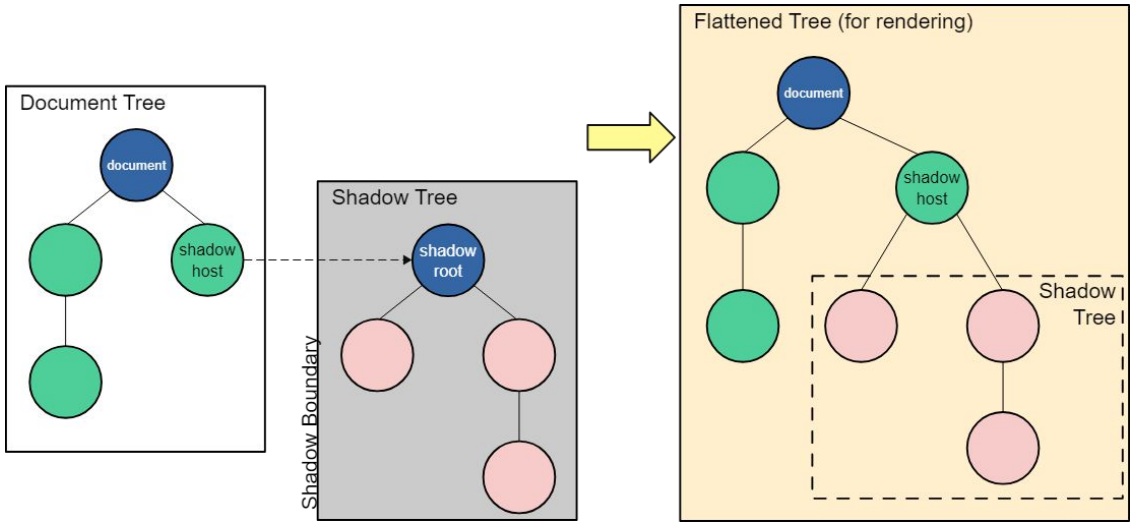
Disadvantage	Description
Code Complexity	JavaScript-based interactions can be harder to read and maintain.
Potential Instability	JavaScript execution depends on browser behavior, which may vary.
Overuse Risk	Excessive reliance on JavaScript may undermine the purpose of WebDriver testing.

Best Practices of using JavaScriptExecutor

1. **Use WebDriver Locators First**:
    a. Rely on WebDriver's native capabilities whenever possible. Use JavaScript Executor as a fallback.
2. **Avoid Overuse**:
    a. Limit JavaScript Executor usage to scenarios where WebDriver methods fail.
3. **Handle Browser Compatibility**:
    a. Ensure JavaScript works across all target browsers.
4. **Reusable Methods**:
    a. Create utility methods for repetitive JavaScript-based interactions.

Working with Shadow DOM

- The **Shadow DOM** is a feature of the Web Components standard that allows encapsulation of HTML, CSS, and JavaScript within a shadow tree. This provides better modularity and prevents styles or scripts from leaking into or out of a component.
- Selenium WebDriver provides two primary methods for handling shadow DOM elements: the modern getShadowRoot() method in Selenium 4.x and the legacy approach using JavaScript Executor.



What is Shadow DOM?

The **Shadow DOM** creates a hidden DOM tree attached to a regular DOM element. This ensures that styles or scripts inside the shadow DOM do not interfere with the rest of the page.

In this example, #shadow-root indicates a shadow DOM that is not accessible directly via traditional WebDriver locators like findElement.

```
<html>
    <body>
        <h1>Main Page</h1>
        <my-component>
            #shadow-root (open)
            <div>
                <input type="text" id="shadowInput" placeholder="Enter text here">
                <button id="submitButton">Submit</button>
            </div>
        </my-component>
    </body>
</html>
```

Challenges with Shadow DOM

1. Shadow DOM isolates its elements, requiring special handling to access them.
2. Standard WebDriver locators do not work for encapsulated elements.
3. Some shadow DOMs may be nested, complicating navigation.

Methods for Handling Shadow DOM

1. Using getShadowRoot() (Selenium 4.x)

Selenium 4.x introduced the getShadowRoot() method, which provides native support for accessing and interacting with shadow DOM elements.

Example:

```
// Locate the shadow host
WebElement shadowHost = driver.findElement(By.tagName("my-component"));

// Use getShadowRoot to retrieve the shadow DOM
SearchContext shadowRoot = shadowHost.getShadowRoot();

// Interact with elements inside the shadow DOM
WebElement inputField = shadowRoot.findElement(By.cssSelector("#shadowInput"));
inputField.sendKeys("Hello, Shadow DOM!");
```

Methods for Handling Shadow DOM Contd.

2. Using JavaScript Executor

For browsers or frameworks where Selenium's native methods may not work, JavaScript Executor can be used to traverse the shadow DOM.

Example:

```
// Locate the shadow host
WebElement shadowHost = driver.findElement(By.tagName("my-component"));

// Use JavaScript to access the shadow root
JavascriptExecutor js = (JavascriptExecutor) driver;
SearchContext shadowRoot = (SearchContext) js.executeScript("return arguments[0].shadowRoot", shadowHost);

// Interact with elements inside the shadow DOM
WebElement inputField = shadowRoot.findElement(By.cssSelector("#shadowInput"));
inputField.sendKeys("Hello, Shadow DOM!");
```

Methods for Handling Shadow DOM Contd.

3. Handling Nested Shadow DOMs

Shadow DOMs can contain additional shadow DOMs within them. This requires sequential access.

Example:

```
<my-component>
    #shadow-root (open)
    <nested-component>
        #shadow-root (open)
        <input type="text" id="nestedInput">
    </nested-component>
</my-component>
```

```
// Access the outer shadow host and shadow root
WebElement outerHost = driver.findElement(By.tagName("my-component"));
SearchContext outerShadowRoot = outerHost.getShadowRoot();

// Access the nested shadow host and shadow root
WebElement nestedHost = outerShadowRoot.findElement(By.tagName("nested-component"));
SearchContext nestedShadowRoot = nestedHost.getShadowRoot();

// Interact with the nested input field
WebElement nestedInput =
nestedShadowRoot.findElement(By.cssSelector("#nestedInput"));
nestedInput.sendKeys("Nested Shadow DOM");
```

Best Practices for handling Shadow DOM

1. **Use Native Support**: Prefer **getShadowRoot()** wherever possible for clean and maintainable code.
2. **Fall Back to JavaScript**: Use JavaScript Executor for complex scenarios or unsupported browsers.
3. **Reusable Methods**: Create utility methods to streamline shadow DOM interactions:

```
public WebElement getShadowElement(WebElement shadowHost, String cssSelector) {
    SearchContext shadowRoot = shadowHost.getShadowRoot();
    return shadowRoot.findElement(By.cssSelector(cssSelector));
}
```

4. **Explicit Waits**: Combine shadow DOM handling with explicit waits for dynamic content.

Exceptions in Selenium WebDriver

- Exceptions in Selenium WebDriver occur when an unexpected event disrupts the normal flow of automation scripts.
- Understanding these exceptions, their causes, and how to handle them effectively is crucial for building robust and maintainable test scripts.

Common Selenium Exceptions:

Exception Name	**Cause**
NoSuchElementException	When the requested element is not found in the DOM.
NoSuchWindowException	When the requested browser window is not found.
NoSuchFrameException	When the requested frame is not found.
StaleElementReferenceException	When the referenced element is no longer attached to the DOM.
ElementNotInteractableException	When an element is present but cannot be interacted with (e.g., hidden).
TimeoutException	When a command exceeds the specified timeout.
ElementClickInterceptedException	When an element is obscured by another element, preventing interaction.
InvalidSelectorException	When an invalid or unsupported selector is used.

Exceptions in Selenium WebDriver Contd.

Common Selenium Exceptions:

Exception Name	**Cause**
WebDriverException	A generic exception for WebDriver-related issues.
SessionNotCreatedException	When the WebDriver session cannot be established.
NoSuchAttributeException	When the requested attribute of an element is not found.
JavascriptException	When there is an error in the JavaScript code execution.
MoveTargetOutOfBoundsException	When attempting to interact with an element outside the viewport.
InvalidArgumentException	When an invalid argument is passed to a WebDriver command.
UnhandledAlertException	When an unexpected alert is present.
ScriptTimeoutException	When a JavaScript execution takes longer than the specified timeout.

Exceptions - Detailed Examples

1. Handling NoSuchElementException

Occurs when the element locator cannot find the element.

Example:

```
try {
    WebElement element = driver.findElement(By.id("nonExistentElement"));
} catch (NoSuchElementException e) {
    System.out.println("Element not found: " + e.getMessage());
}
```

2. Handling StaleElementReferenceException

Occurs when the DOM is updated and the previously located element becomes invalid.

Example:

```
try {
    WebElement element = driver.findElement(By.id("dynamicElement"));
    element.click();
    driver.navigate().refresh();
    element.click(); // Causes StaleElementReferenceException
} catch (StaleElementReferenceException e) {
    WebElement refreshedElement =
driver.findElement(By.id("dynamicElement"));
    refreshedElement.click();
}
```

Exceptions - Detailed Examples Contd.

3. Handling ElementNotInteractableException

Occurs when an element is present but not interactable (e.g., hidden).

Example:

```
try {
    WebElement hiddenButton = driver.findElement(By.id("hiddenButton"));
    hiddenButton.click();
} catch (ElementNotInteractableException e) {
    System.out.println("Element not interactable: " + e.getMessage());
}
```

4. Handling TimeoutException

Occurs when an operation takes longer than the specified wait time.

Example:

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
try {
    WebElement element =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("dynamicElement")));
} catch (TimeoutException e) {
    System.out.println("Timeout while waiting for element: " + e.getMessage());
}
```

Exceptions - Detailed Examples Contd.

5. Handling UnhandledAlertException

Occurs when an unexpected alert is present.

Example:

```
try {
    driver.findElement(By.id("triggerAlert")).click();
} catch (UnhandledAlertException e) {
    Alert alert = driver.switchTo().alert();
    alert.accept();
}
```

Exceptions - Detailed Examples Contd.

Custom Exception Handling Example

Implementing a reusable utility for element interaction:

```
public WebElement safeFindElement(By locator, int timeout) {
    try {
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(timeout));
        return wait.until(ExpectedConditions.presenceOfElementLocated(locator));
    } catch (NoSuchElementException e) {
        System.out.println("Element not found: " + locator);
        throw e;
    } catch (TimeoutException e) {
        System.out.println("Timeout waiting for element: " + locator);
        throw e;
    }
}
```

Taking Screenshots in Selenium WebDriver

- Capturing screenshots during test execution is a critical feature for debugging, reporting, and verifying UI elements.
- Selenium WebDriver provides multiple ways to take screenshots, including full-page and element-specific captures.

Why Take Screenshots?

1. **Debugging Failures**: Visual evidence helps identify issues during test failures.
2. **Reporting**: Enhances the clarity of test reports with visual proof.
3. **Verification**: Validates UI layouts, content, and dynamic behavior.

Methods to Take Screenshots:

Method	Description
TakesScreenshot interface	Captures the visible portion of the web page.
getScreenshotAs()	Captures screenshots of specific elements, full pages, or the current viewport.
3rd-party libraries (e.g., AShot)	Useful for advanced use cases like capturing tooltips, overlays, or stitching screenshots.

Taking Screenshots

1. Taking a Page Screenshot (Selenium 4.x)

Syntax:

```
File screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
FileUtils.copyFile(screenshot, new File("screenshots/fullPage.png"));
```

2. Taking a Screenshot of a Web Element

In Selenium 4.x, you can capture specific web elements directly.

Syntax:

```
WebElement element = driver.findElement(By.id("logo"));
File elementScreenshot = element.getScreenshotAs(OutputType.FILE);
FileUtils.copyFile(elementScreenshot, new File("screenshots/element.png"));
```

Taking Screenshots Contd.

3. Taking a Screenshot as a Base64 String

You can also capture a screenshot and encode it in Base64, useful for embedding directly in reports.

Syntax:

```
String base64Screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.BASE64);
// Use this base64 string for embedding in reports.
System.out.println("Base64 Screenshot: " + base64Screenshot);
```

4. Capturing Full Page Screenshots

Selenium 4.x supports full-page screenshots for certain browsers (e.g., Firefox).

Syntax:

```
byte[] imageBytes = ((FirefoxDriver) driver).getFullPageScreenshotAs(OutputType.BYTES);
Path destination = Paths.get("fullpage-screenshot-firefox.png");
Files.write(destination, imageBytes);
```

Taking Screenshots Contd.

Utility Method for Taking Screenshots

```
public void takeScreenshot(String fileName) {
    try {
        File screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
        FileUtils.copyFile(screenshot, new File("screenshots/" + fileName + ".png"));
        System.out.println("Screenshot saved as " + fileName + ".png");
    } catch (IOException e) {
        System.out.println("Failed to take screenshot: " + e.getMessage());
    }
}
```

Taking Screenshots Contd.

Comparison of Output Types

OutputType	Description	Use Case
FILE	Saves the screenshot as a file.	Suitable for local storage.
BASE64	Encodes the screenshot as a string.	Embedding in logs or reports.
BYTES	Returns the screenshot as a byte[] array.	Used for advanced processing or APIs.

Taking Screenshots - Best Practices

1. **Organize Screenshots**:
    a. Save screenshots in a structured folder hierarchy (e.g., by test case or execution date).
2. **Capture Screenshots on Failures**:
    a. Integrate screenshot capture with test frameworks like TestNG or JUnit for failed tests.
3. **Optimize Performance**:
    a. Avoid overusing screenshots to prevent excessive file generation and slower execution.
4. **Cross-Browser Considerations**:
    a. Validate screenshot capabilities across browsers as some may not fully support features like full-page screenshots.

WebDriver Architecture Overview

Selenium WebDriver uses a client-server architecture where commands from test scripts are sent to the browser-specific driver, which executes them and returns the response.

Key Components:

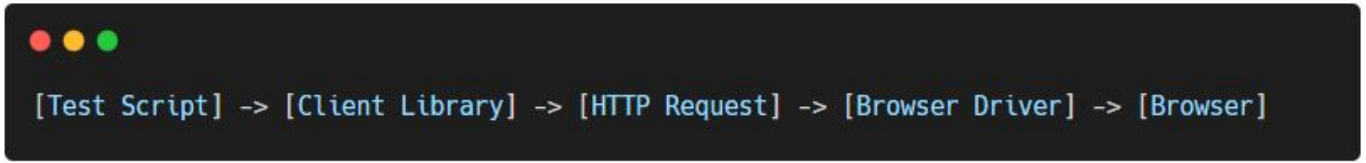
Component	Description
1. Client Library	Language-specific bindings (Java, Python, C#, etc.) to interact with WebDriver.
2. JSON Wire Protocol / W3C Protocol	Standard protocol for communication between client and browser driver.
3. Browser-Specific Driver	Executable that translates WebDriver commands to browser actions.
4. Browser	The web browser where test actions are executed.

WebDriver Working Process

1. **Test Script Execution**:
    - Test scripts use Selenium language bindings to send commands like opening a URL, clicking, or typing text.
2. **Command Translation**:
    - The Selenium bindings convert these commands into HTTP requests using the JSON Wire Protocol or W3C WebDriver Protocol (v4.x onwards).
3. **Driver Communication**:
    - The HTTP requests are sent to the browser-specific driver (e.g., chromedriver, geckodriver).
    - The driver acts as a middleman, interpreting WebDriver commands into browser-specific actions.
4. **Command Execution in the Browser**:
    - The browser executes the requested actions and sends the response back to the driver.
5. **Response to the Client**:
    - The driver sends the browser's response to the WebDriver client library, which the test framework uses.

WebDriver Working Process - Detailed Workflow

Step	Example	Details
Step 1: Test Script	driver.get("http://example.com");	User writes a test script using Selenium language bindings.
Step 2: Command Serialization	Converts the command into HTTP request.	POST /session/{session_id}/url with URL payload.
Step 3: Send to Driver	Sent to chromedriver via HTTP.	The browser-specific driver receives and interprets the command.
Step 4: Command Execution	Browser navigates to the URL.	The browser executes the corresponding action.
Step 5: Response	Success or failure response is returned.	Driver returns an HTTP response to the WebDriver client.



W3C WebDriver Protocol

Selenium 4.x exclusively supports the W3C WebDriver protocol, which improves compatibility and performance by standardizing WebDriver commands across browsers.

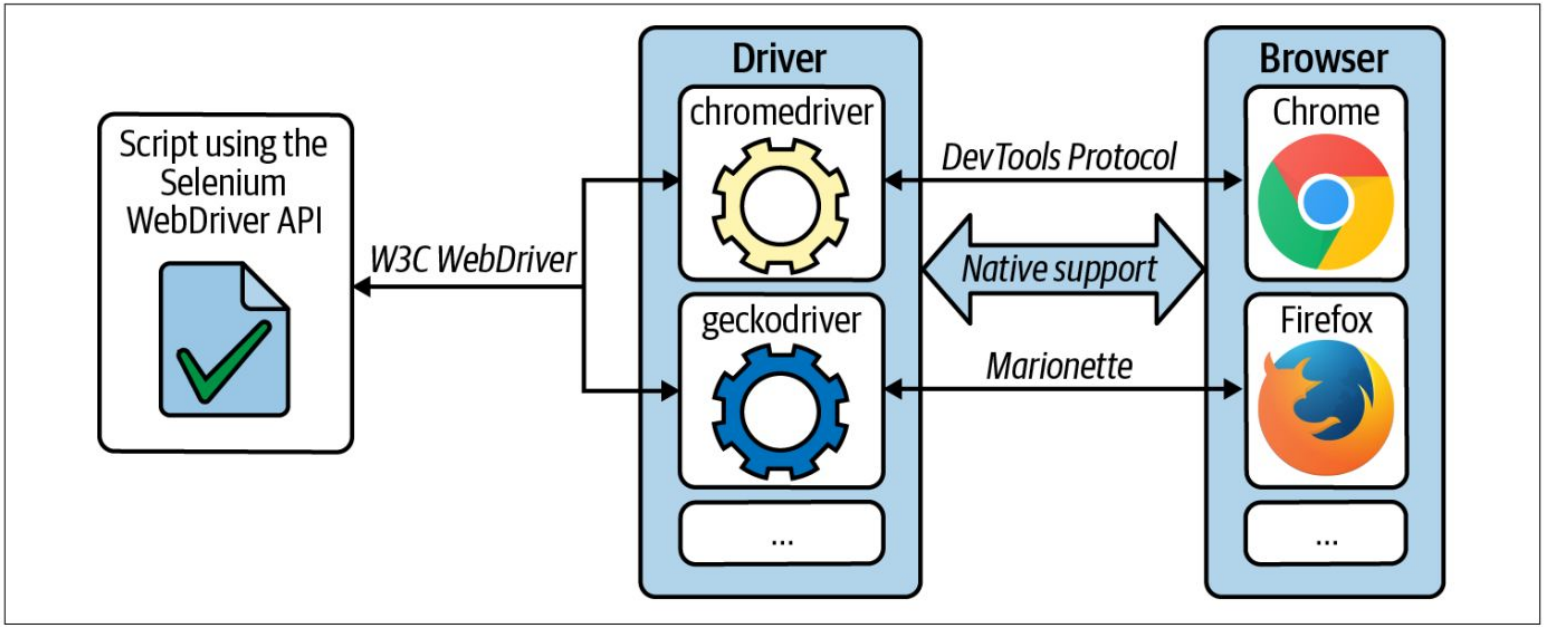
Key Features:

- Reduced communication overhead.
- Enhanced cross-browser compatibility.
- Support for advanced interactions like file uploads and custom scripts.

Browser-Specific Drivers:

Driver	Supported Browser	Notes
chromedriver	Google Chrome	Provided by Chrome; works with Chrome browser versions.
geckodriver	Mozilla Firefox	Supported for Firefox; adheres to W3C WebDriver.
msedgedriver	Microsoft Edge	Supported for Edge; works seamlessly with Edge browser.
iedriverserver	Internet Explorer	Deprecated in favor of modern browsers like Edge.
safaridriver	Safari (macOS)	Built into Safari; requires enabling remote automation.

WebDriver Architecture



WebDriver Major Classes and Interfaces

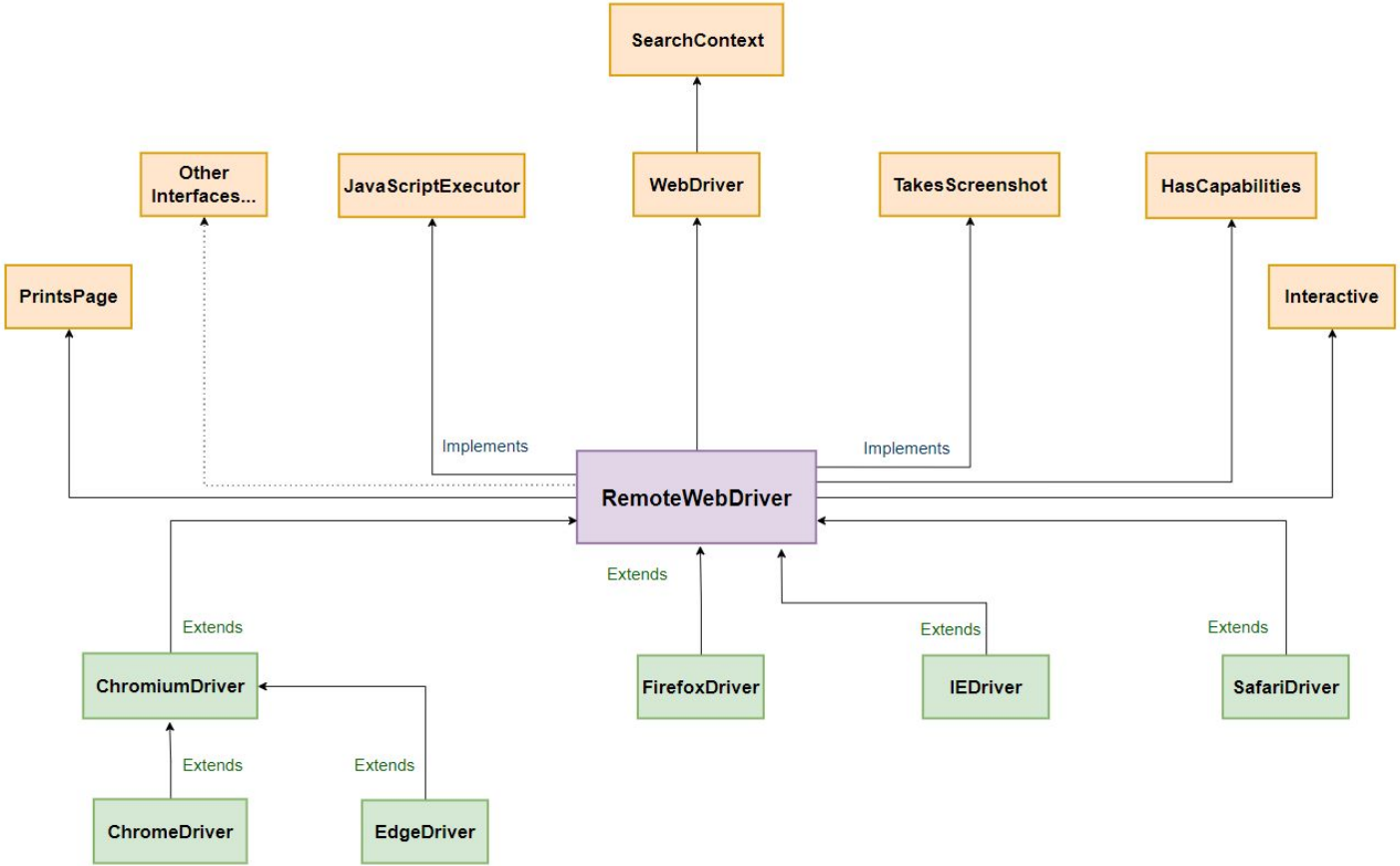
- In Selenium WebDriver, various classes and interfaces play key roles in interacting with browsers, web elements, and executing automation scripts.
- Understanding these core components helps in effectively leveraging WebDriver's capabilities.

Type	Name	Purpose	Key Methods/Features	Example
Interface	**WebDriver**	Central interface for browser control.	get(), getTitle(), getCurrentUrl(), quit()	WebDriver driver = new ChromeDriver();
	WebElement	Represents elements on a web page.	click(), sendKeys(), getText(), isDisplayed()	WebElement element = driver.findElement(By.id("example"));
	TakesScreenshot	Captures screenshots of web pages.	getScreenshotAs()	File src = ((TakesScreenshot)driver).getScreenshotAs(...);
	JavascriptExecutor	Executes JavaScript within the browser.	executeScript(), executeAsyncScript()	((JavascriptExecutor)driver).executeScript("alert('Hi');");
	Alert	Handles JavaScript alerts, prompts, and confirmation boxes.	accept(), dismiss(), getText(), sendKeys()	Alert alert = driver.switchTo().alert();

WebDriver Major Classes and Interfaces Contd.

Type	Name	Purpose	Key Methods/Features	Example
Class	**RemoteWebDriver**	Parent class for all browser-specific drivers.	Inherits methods from WebDriver.	WebDriver driver = new RemoteWebDriver(...);
	ChromeDriver	Controls Google Chrome browser.	Browser-specific implementation.	WebDriver driver = new ChromeDriver();
	FirefoxDriver	Controls Mozilla Firefox browser.	Browser-specific implementation.	WebDriver driver = new FirefoxDriver();
	EdgeDriver	Controls Microsoft Edge browser.	Browser-specific implementation.	WebDriver driver = new EdgeDriver();
	SafariDriver	Controls Safari browser on macOS.	Browser-specific implementation.	WebDriver driver = new SafariDriver();
Utility Class	**Actions**	Handles advanced user interactions (e.g., drag-and-drop, key presses).	moveToElement(), click(), dragAndDrop()	Actions actions = new Actions(driver);
	Select	Simplifies interaction with dropdown menus.	selectByVisibleText(), getOptions(), deselectAll()	Select dropdown = new Select(driver.findElement(By.id("menu")));
	FluentWait	Provides flexible wait with timeout and polling.	withTimeout(), pollingEvery(), until()	FluentWait<WebDriver> wait = new FluentWait<>(driver)...;

WebDriver Major Classes and Interfaces Hierarchy



Browser Drivers & Browser-Specific Functionality

What are Browser Drivers?

- **Purpose**: Browser drivers are executables provided by browser vendors to interact with the browser via WebDriver protocols.
- **Role**: They translate WebDriver commands into browser-native actions.
- **Examples**:
    - ChromeDriver for Google Chrome.
    - GeckoDriver for Mozilla Firefox.
    - EdgeDriver for Microsoft Edge.
    - SafariDriver for Safari.

Common Features Across Browser Drivers

- **W3C WebDriver Protocol Compliance**: All modern drivers support the W3C WebDriver standard for cross-browser compatibility.
- **Parallel Execution Support**: Allows running multiple browser instances in parallel.
- **Integration with Browser-Specific Tools**: Some drivers offer access to browser developer tools for enhanced testing.

Browser-Specific Drivers and Functionalities

Browser	Driver	Key Features	Specific Functionalities
Google Chrome	ChromeDriver	- Supports latest Chrome versions. - Integrates with Chrome DevTools for debugging.	- Access network logs using DevTools Protocol. - Emulate devices (e.g., smartphones) with device metrics. - Automate downloads and uploads.
Mozilla Firefox	GeckoDriver	- Supports Firefox Quantum and later versions. - Independent driver compliant with W3C specs.	- Headless mode for faster execution. - Interact with browser profiles for custom settings. - Use about:config settings for advanced configurations.
Microsoft Edge	EdgeDriver	- Supports Chromium-based Edge versions. - Integrates seamlessly with Windows environments.	- Use Chromium features like DevTools Protocol. - Automate scenarios specific to Windows environments (e.g., authentication dialogs).
Apple Safari	SafariDriver	- Built into macOS (no separate installation required). - Strict security sandboxing.	- Works only on macOS. - Supports native AppleScript for advanced automation. - Limited debugging and inspection tools compared to other browsers.
Opera	Operadriver	- Similar to ChromeDriver (since Opera is based on Chromium).	- Supports custom Opera settings. - Provides unique UI-related test scenarios.

Browser Drivers - Customization Using Options Classes

What are Options Classes?

Options classes are specialized classes provided for each browser to set configurations before initializing the WebDriver instance. These configurations include:

- Adding command-line arguments.
- Setting browser preferences.
- Disabling browser-specific features like popup blocking.
- Configuring proxies or SSL handling.

Options Classes for Different Browsers:

Browser	Options Class	Usage	Example Usage
Google Chrome	**ChromeOptions**	Configure ChromeDriver with arguments, extensions, and experimental options.	`ChromeOptions options = new ChromeOptions();`
Mozilla Firefox	**FirefoxOptions**	Set Firefox-specific preferences, profiles, and arguments.	`FirefoxOptions options = new FirefoxOptions();`
Microsoft Edge	**EdgeOptions**	Add Chromium-based Edge configurations.	`EdgeOptions options = new EdgeOptions();`
Safari	**SafariOptions**	Configure SafariDriver with platform-specific options.	`SafariOptions options = new SafariOptions();`
Opera	**ChromeOptions (same)**	Opera is based on Chromium, so `ChromeOptions` is used.	`ChromeOptions options = new ChromeOptions();`

Common Configurations Using Options Classes

1. Headless Mode

Run the browser without a GUI for faster execution.

Example (Chrome):

```
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless");
WebDriver driver = new ChromeDriver(options);
```

2. Adding Extensions

Install browser extensions during automation.

Example (Firefox):

```
FirefoxOptions options = new FirefoxOptions();
options.addExtensions(new File("path/to/extension.xpi"));
WebDriver driver = new FirefoxDriver(options);
```

3. **Disabling Popup Blocking**

Prevent browsers from blocking popups during testing.

Example (Chrome):

```
ChromeOptions options = new ChromeOptions();
options.addArguments("--disable-popup-blocking");
WebDriver driver = new ChromeDriver(options);
```

4. Setting Browser-Specific Preferences

Custom preferences like download directories or security options.

Example (Firefox):

```
FirefoxOptions options = new FirefoxOptions();
options.addPreference("browser.download.dir", "C:\\Downloads");
options.addPreference("browser.download.folderList", 2);
WebDriver driver = new FirefoxDriver(options);
```

Common Configurations Using Options Classes Contd.

5. Proxy Configuration

Set up a proxy for testing web traffic.

Example (Chrome):

```
ChromeOptions options = new ChromeOptions();
options.addArguments("--proxy-server=http://proxyserver:8080");
WebDriver driver = new ChromeDriver(options);
```

6. Disabling Notifications

Suppress browser notifications during tests.

Example (Chrome):

```
ChromeOptions options = new ChromeOptions();
options.addArguments("--disable-notifications");
WebDriver driver = new ChromeDriver(options);
```

Options Class Usage Summary

Configuration	Browser	Options Class	Example Code
Headless Mode	Chrome, Firefox	ChromeOptions, FirefoxOptions	options.addArguments("--headless");
Disabling Notifications	Chrome, Firefox	ChromeOptions, FirefoxOptions	options.addArguments("--disable-notifications");
Proxy Configuration	All	All Options Classes	options.addArguments("--proxy-server=http://proxyserver:8080");
Adding Extensions	Chrome, Firefox	ChromeOptions, FirefoxOptions	options.addExtensions(new File("path/to/extension"));
Browser-Specific Preferences	Firefox	FirefoxOptions	options.addPreference("browser.download.dir", "C:\\Downloads");
Incognito/InPrivate Mode	Chrome, Edge	ChromeOptions, EdgeOptions	options.addArguments("--incognito");

Cross-Browser Testing with Selenium WebDriver

What is Cross-Browser Testing?

- **Definition**: Testing a web application on multiple browser environments to verify its functionality, performance, and user experience.
- **Purpose**:
  - Identify browser-specific issues.
  - Ensure compatibility with older and modern browser versions.
  - Validate responsive design across devices.

Importance of Cross-Browser Testing:

Reason	Description
Browser Diversity	Users access websites using various browsers like Chrome, Firefox, Safari, etc.
Rendering Differences	Browsers render HTML, CSS, and JavaScript differently.
User Experience Consistency	Deliver a uniform experience to all users.
Increased Test Coverage	Ensure your application works seamlessly for a wider audience.

Key Components of Cross-Browser Testing

Component	Description
Browser Drivers	Required to interact with specific browsers (e.g., ChromeDriver, GeckoDriver).
Options Classes	Allow browser-specific configurations for flexibility.
Grid Integration	Distribute tests across multiple environments in parallel.

Steps for Cross-Browser Testing

Step 1: Define Supported Browsers

- Identify browsers and versions based on your audience (e.g., Chrome 110+, Firefox 102+, Edge 110+, etc.).

Step 2: Set Up WebDriver for Multiple Browsers

- Configure WebDriver for each target browser.

Step 3: Design Test Framework

- Write reusable and modular test scripts to adapt across browsers.

Step 4: Parameterize Browser Configurations

- Pass the browser type dynamically through configuration files or environment variables.

Step 5: Execute Tests

- Run tests sequentially or in parallel for efficiency.

Dynamic Browser Initialization

Dynamic initialization ensures the same test script can work across different browsers:

Example:

```
public WebDriver initializeDriver(String browser) {
    WebDriver driver;
    if (browser.equalsIgnoreCase("chrome")) {
        ChromeOptions options = new ChromeOptions();
        driver = new ChromeDriver(options);
    } else if (browser.equalsIgnoreCase("firefox")) {
        FirefoxOptions options = new FirefoxOptions();
        driver = new FirefoxDriver(options);
    } else if (browser.equalsIgnoreCase("edge")) {
        EdgeOptions options = new EdgeOptions();
        driver = new EdgeDriver(options);
    } else {
        throw new IllegalArgumentException("Unsupported browser: " + browser);
    }
    return driver;
}
```

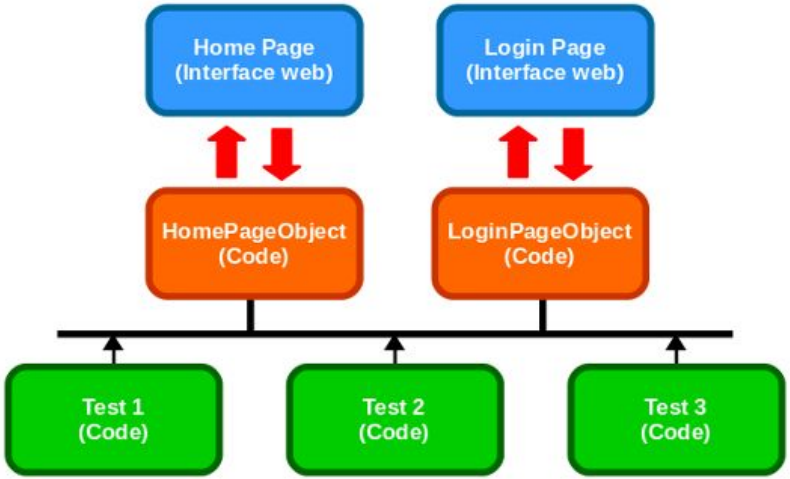
Test Execution:

```
String browser = System.getProperty("browser", "chrome");
WebDriver driver = initializeDriver(browser);
```

Page Object Model (POM) Approach in Selenium

What is Page Object Model (POM)?

- **Definition**: POM is a framework design approach where every web page in the application is represented as a separate class.
- **Structure**:
  - **Web Elements**: Represented as variables.
  - **Page Methods**: Represent user interactions like clicking a button, entering text, etc.
- **Goal**: Create a clean separation between test logic and UI structure.



Page Object Model (POM) - Pros & Cons

Advantages of POM:

Advantages	Description
Improves Maintainability	Centralized page elements make updates easier if the UI changes.
Enhances Code Reusability	Methods for actions like login can be reused across multiple test cases.
Reduces Code Duplication	Web elements and methods are defined once and used throughout.
Increases Readability	Logical structure makes it easier for teams to understand and modify.
Supports Modularity	Each page has its own class, improving the organization of test scripts.

Disadvantages of POM:

Disadvantage	Description
Initial Setup Overhead	Designing and creating POM structure requires more effort upfront.
Increased Complexity	For large applications, maintaining multiple page classes can become complex.
Dependency on Locators	Frequent UI changes may still require updates in locators.

Code Example - Page Object Model (POM)

```java
public class LoginPage {
    // Locators
    By usernameField = By.id("username");
    By passwordField = By.id("password");
    By loginButton = By.id("login");

    WebDriver driver;

    // Constructor
    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    // Actions
    public void enterUsername(String username) {
        driver.findElement(usernameField).sendKeys(username);
    }

    public void enterPassword(String password) {
        driver.findElement(passwordField).sendKeys(password);
    }

    public void clickLogin() {
        driver.findElement(loginButton).click();
    }
}
```

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class LoginTest {

    private WebDriver driver;
    private LoginPage loginPage;

    @BeforeClass
    public void setUp() {
        driver = new ChromeDriver();
        driver.get("https://example.com/login");

        // Initialize the LoginPage object
        loginPage = new LoginPage(driver);
    }

    @Test
    public void testLogin() {
        // Perform login
        loginPage.enterUsername("testUser");
        loginPage.enterPassword("testPassword");
        loginPage.clickLogin();

        // Add an assertion (example: check for a successful login message)
        String expectedTitle = "Welcome Page";
        String actualTitle = driver.getTitle();
        assert actualTitle.equals(expectedTitle) : "Login test failed!";
    }

    @AfterClass
    public void tearDown() {
        // Close the browser
        if (driver != null) {
            driver.quit();
        }
    }
}
```

Page Factory in POM

The **Page Factory** is an extension of the POM design pattern that simplifies the initialization of web elements using the @FindBy annotation.

Key Features of Page Factory:

1. Uses @FindBy to locate elements.
2. Automatically initializes web elements using initElements() method.
3. Reduces boilerplate code for web element declarations.

Code Example - Page Object Model (POM) with Page Factory

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class LoginPage {
    WebDriver driver;

    // Locators with @FindBy
    @FindBy(id = "username")
    WebElement usernameField;

    @FindBy(id = "password")
    WebElement passwordField;

    @FindBy(id = "login")
    WebElement loginButton;

    // Constructor
    public LoginPage(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }

    // Actions
    public void enterUsername(String username) {
        usernameField.sendKeys(username);
    }

    public void enterPassword(String password) {
        passwordField.sendKeys(password);
    }

    public void clickLogin() {
        loginButton.click();
    }
}
```

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class LoginTest {

    private WebDriver driver;
    private LoginPage loginPage;

    @BeforeClass
    public void setUp() {
        driver = new ChromeDriver();
        driver.get("https://example.com/login");

        // Initialize the LoginPage object
        loginPage = new LoginPage(driver);
    }

    @Test
    public void testLogin() {
        // Perform login
        loginPage.enterUsername("testUser");
        loginPage.enterPassword("testPassword");
        loginPage.clickLogin();

        // Add an assertion (example: check for a successful login message)
        String expectedTitle = "Welcome Page";
        String actualTitle = driver.getTitle();
        assert actualTitle.equals(expectedTitle) : "Login test failed!";
    }

    @AfterClass
    public void tearDown() {
        // Close the browser
        if (driver != null) {
            driver.quit();
        }
    }
}
```

Differences: POM vs. Page Factory

Aspect	Plain Page Object Model (POM)	POM with Page Factory
Web Element Declaration	Web elements are manually declared and located using By or WebElement.	Web elements are annotated with @FindBy.
Element Initialization	Manually located each time using driver.findElement().	Automatically initialized using PageFactory.initElements().
Code Length	Requires more boilerplate code for locating elements.	Reduces boilerplate code with concise annotations.
Performance	Slightly better performance as it avoids lazy initialization overhead.	May lead to stale element exceptions due to lazy initialization.
Error Handling	Offers better control to handle exceptions like NoSuchElementException.	Prone to StaleElementReferenceException if the DOM changes.
Dynamic Element Handling	Suited for dynamically changing web elements with frequent updates.	Not ideal for dynamic elements due to lazy initialization.
Readability	May require more lines of code but provides better visibility for element locators.	More concise and readable due to @FindBy annotations.
Ease of Maintenance	Requires manual updates to element locators in the class.	Centralizes element initialization but may be harder to debug issues caused by lazy loading.
Usage	Preferred for applications with highly dynamic or complex pages.	Suitable for moderately complex applications with stable page structures.
Flexibility	Can be customized to handle specific scenarios, including dynamic waits.	Limited flexibility due to strict reliance on annotations.
Recommended By Selenium Authors	Yes, as it provides more direct control over WebDriver interactions.	No longer recommended due to stale element issues and lack of control.

Best Practices for using POM

1. **Avoid Mixing Logic**: Keep test logic separate from page classes.
2. **Use Meaningful Names**: Name web elements and methods clearly to improve readability.
3. **Encapsulate Actions**: Provide high-level methods (e.g., login() instead of separate enterUsername() and enterPassword() methods).
4. **Optimize Locator Strategy**: Use efficient locator strategies like id and CSS selectors.

Selenium Grid

What is Selenium Grid?

- **Definition**: A component of the Selenium suite designed for distributed and parallel testing.
- **Purpose**:
  - Execute tests across different environments and browsers.
  - Improve test execution time by running tests concurrently.
  - Centralize test execution management.
- **Key Components**:
  - **Hub**: The central point that receives test execution requests and distributes them to Nodes.
  - **Nodes**: Machines (physical or virtual) that execute test scripts using specified browser configurations.

Enhancements in Selenium Grid 4.x:

1. **Simplified Architecture**:
   - Unified configuration, replacing the traditional Hub-Node architecture.
   - Support for **Standalone**, **Hub-Node**, and **Distributed** modes.
2. **Docker Support**: Seamless integration with Docker containers for scalable test execution.
3. **Built-in Observability**:
   - Tracks test executions with better logs and metrics.
4. **Supports W3C WebDriver Protocol**: Fully compliant with modern WebDriver standards.

Selenium Grid Architecture

Modes of Operation:

1. **Standalone Mode**:
    - Suitable for small-scale testing.
    - Single instance acts as both Hub and Node.
2. **Hub-Node Mode**:
    - Traditional configuration with a central Hub and multiple Nodes.
3. **Distributed Mode**:
    - Scalable architecture with separate components (Hub, Distributor, Event Bus, Nodes, and Session Map) for large-scale testing.

Distributed Mode Components:

Component	**Role**
Hub	Entry point for test requests, handles session creation.
Distributor	Assigns test sessions to Nodes based on capabilities.
Event Bus	Facilitates communication between components.
Session Map	Keeps track of active test sessions.
Node	Executes the test cases using browsers and drivers.

Installing & Configuring Selenium Grid

Installation Steps:

1. **Download Selenium Server**:
   - Download the .jar file from Selenium's official site.
2. **Install Java**:
   - Ensure Java is installed and properly configured.

Configuration Options:

1. **Standalone Mode**:

   java -jar selenium-server-<version>.jar standalone

2. **Hub-Node Mode**:

   Start Hub:
   java -jar selenium-server-<version>.jar hub

   Start Node:
   java -jar selenium-server-<version>.jar node --hub http://localhost:4444

3. **Distributed Mode**:
   - Start Components Individually (Hub, Distributor, Event Bus, Nodes, etc.).

Selenium Grid - Remote Test Execution

Setting Up Remote Execution:

1. **Start Selenium Grid**: Use one of the above configurations to launch the Grid.
2. **Test Configuration**:
    - Update WebDriver URL to point to the Hub:

```
WebDriver driver = new RemoteWebDriver(
    new URL("http://localhost:4444/wd/hub"),
    new ChromeOptions()
);
```

3. **Run Tests**: Execute test scripts, and the Grid will distribute them to appropriate Nodes.

Selenium Grid - Parallel Testing

Benefits:

- Reduces total execution time.
- Increases test coverage across multiple browsers and environments.

Implementation:

- Use **TestNG** or **JUnit** to configure parallel testing.

Example (TestNG XML Configuration):

```
<suite name="Parallel Tests" parallel="tests" thread-count="3">
    <test name="Test1">
        <classes>
            <class name="tests.ChromeTest"/>
        </classes>
    </test>
    <test name="Test2">
        <classes>
            <class name="tests.FirefoxTest"/>
        </classes>
    </test>
</suite>
```

Selenium WebDriver Best Practices and Tips

1. General Best Practices

- Follow the **DRY Principle**: Create reusable methods for repetitive actions.
- Use the **Page Object Model (POM)** to separate test logic from page interactions.
- Replace **implicit waits** with **explicit waits** for better control over synchronization.
- Avoid **hard-coded values** by storing test data, URLs, and credentials in configuration files or environment variables.
- Always **close resources properly** by quitting the WebDriver instance in the teardown step.

2. Locator Strategies

- Prefer **unique locators** like IDs for faster and more stable identification.
- Avoid **absolute XPaths**; use relative XPaths to make tests more robust against DOM changes.
- Leverage **CSS selectors** for speed and simplicity when possible.
- Test and validate locators in the **browser console** before adding them to your scripts.

Selenium WebDriver Best Practices and Tips Contd.

3. Synchronization

- Use **explicit waits** with conditions like visibilityOf, elementToBeClickable, or presenceOfElementLocated.
- Avoid using Thread.sleep() as it introduces fixed delays and reduces test reliability.
- Handle **dynamic elements** with custom wait methods tailored to your application’s behavior.

4. Coding Practices

- Write **modular tests** by breaking them into smaller, independent units for easier maintenance.
- Use **meaningful assertions** to validate expected outcomes clearly.
- Implement **logging** to track key actions (e.g., navigation, clicks, validations) for better debugging.
- Include **exception handling** with descriptive messages for easier troubleshooting.

5. Cross-Browser Testing

- Test on **major browsers** like Chrome, Firefox, Edge, and Safari based on your application's audience.
- Customize WebDriver instances with **browser options classes** for scenarios like headless testing or disabling pop-ups.
- Use frameworks like **TestNG** or **JUnit** to run tests in parallel across multiple browsers.

Selenium WebDriver Best Practices and Tips Contd.

6. Handling Flaky Tests

- Analyze failures to identify **root causes** (e.g., synchronization issues, unstable environments).
- Implement **retry logic** to rerun intermittently failing tests.
- Ensure tests are **independent** and do not rely on the execution order or shared states.
- Use tools like **Docker** to stabilize the test environment and avoid inconsistencies.

7. Debugging and Reporting

- Capture **screenshots on test failures** for detailed analysis.
- Use reporting tools like **Allure** or **Extent Reports** to generate detailed test execution summaries.
- Log and monitor **browser console errors** to identify JavaScript or other browser-side issues.

Selenium WebDriver Best Practices and Tips Contd.

8. Scalability

- Use **Selenium Grid** or cloud-based platforms like **BrowserStack** or **Sauce Labs** to scale test execution.
- Parameterize tests using tools like **TestNG Data Providers** for data-driven testing.
- Regularly **review and optimize** the test suite to remove redundant or obsolete tests.

9. Advanced Techniques

- Use **JavaScript Executor** for advanced actions like scrolling, injecting scripts, or handling shadow DOM elements.
- Consider **headless browser testing** for faster test execution without rendering the UI.
- Integrate tests into **CI/CD pipelines** (e.g., Jenkins, GitHub Actions, Azure DevOps) for continuous testing and deployment.